FUJITSU MICROELECTRONICS
CONTROLLER MANUAL

CM71-00105-1E

FR81 Family

32-BIT MICROCONTROLLER

PROGRAMMING MANUAL

FUJITSU

FR81 Family

32-BIT MICROCONTROLLER

PROGRAMMING MANUAL

For the information for microcontroller supports, see the following web site.

http://edevice.fujitsu.com/micom/en-support/

FUJITSU MICROELECTRONICS LIMITED

PREFACE

■ Objectives and targeted reader

FR81 Family is a 32 bit single chip microcontroller with CPU of new RISC Architecture as the core. FR81 Family has specifications that are optimum for embedded use requiring high performance CPU processing power.

This manual explains the programming model and execution instructions for engineers developing a product using this FR81 Family Microcontroller, especially the programmers who produce programs using assembly language of the assembler for FR/FR80/FR81 Family.

For the rules of assembly grammar language and the method of use of Assembler Programs, kindly refer to "FR Family Assembler Manual".

*: FR, the abbreviation of Fujitsu RISC controller, is a line of products of Fujitsu Microelectronics Limited.

Other company names and brand names are the trademarks or registered trademarks of their respective owners.

■ Organization of this Manual

This manual consists of the following 7 chapters and 1 supplement.

CHAPTER 1 OVERVIEW OF FR81 FAMILY CPU

This chapter describes the features of FR81 Family CPU and its differences from hitherto FR Family.

CHAPTER 2 MEMORY ARCHITECTURE

This chapter describes Memory Architecture of the CPU of FR81 Family. Memory Architecture is the method of allocation of memory space and access to this memory space.

CHAPTER 3 PROGRAMMING MODEL

This chapter describes registers in the CPU existing as programming model of FR81 Family CPU.

CHAPTER 4 RESET AND "EIT" PROCESSING

This chapter describes resetting of FR81 Family CPU and EIT processing. EIT processing is the generic term for exceptions, interruption and trap.

CHAPTER 5 PIPELINE OPERATION

This chapter describes pipeline operation and delay divergence, the salient feature of FR81 Family CPU.

CHAPTER 6 INSTRUCTION OVERVIEW

This chapter describes outline of commands of FR81 Family CPU.

CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS

This chapter describes Execution Instructions of FR81 Family CPU in Reference Format in the alphabetical order.

APPENDIX

It contains instruction list and instruction map of FR81 Family CPU.

- The contents of this document are subject to change without notice.
  Customers are advised to consult with sales representatives before ordering.
- The information, such as descriptions of function and application circuit examples, in this document are presented solely for the purpose of reference to show examples of operations and uses of FUJITSU MICROELECTRONICS device; FUJITSU MICROELECTRONICS does not warrant proper operation of the device with respect to use based on such information. When you develop equipment incorporating the device based on such information, you must assume any responsibility arising out of such use of the information. FUJITSU MICROELECTRONICS assumes no liability for any damages whatsoever arising out of the use of the information.
- Any information in this document, including descriptions of function and schematic diagrams, shall not be construed as license of the use or exercise of any intellectual property right, such as patent right or copyright, or any other right of FUJITSU MICROELECTRONICS or any third party or does FUJITSU MICROELECTRONICS warrant non-infringement of any third-party's intellectual property right or other right by using such information. FUJITSU MICROELECTRONICS assumes no liability for any infringement of the intellectual property rights or other rights of third parties which would result from the use of information contained herein.
- The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for use requiring extremely high reliability (i.e., submersible repeater and artificial satellite).
  Please note that FUJITSU MICROELECTRONICS will not be liable against you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products.
- Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
- Exportation/release of any products described in this document may require necessary procedures in accordance with the regulations of the Foreign Exchange and Foreign Trade Control Law of Japan and/or US export control laws.
- The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

Copyright ©2009 FUJITSU MICROELECTRONICS LIMITED All rights reserved.

CONTENTS

CHAPTER 1 OVERVIEW OF FR81 FAMILY CPU 1

1.1 Features of FR81 Family CPU 2
1.2 Changes from the earlier FR Family 4

CHAPTER 2 MEMORY ARCHITECTURE 7

2.1 Address Space 8
2.1.1 Direct Address Area 8
2.1.2 Vector Table Area 9
2.1.3 20-bit Addressing Area & 32-bit Addressing Area 11
2.2 Data Structure 12
2.2.1 Byte Data 12
2.2.2 Half Word Data 12
2.2.3 Word Data 12
2.2.4 Byte Order 13
2.3 Word Alignment 14
2.3.1 Program Access 14
2.3.2 Data Access 14

CHAPTER 3 PROGRAMMING MODEL 15

3.1 Register Configuration 16
3.2 General-purpose Registers 17
3.2.1 Configuration of General-purpose Registers 17
3.2.2 Special Usage of General-purpose Registers 18
3.2.3 Relation between Stack Pointer and R15 18
3.3 Dedicated Registers 19
3.3.1 Configuration of Dedicated Registers 19
3.3.2 Program Counter (PC) 20
3.3.3 Program Status (PS) 20
3.3.4 System Status Register (SSR) 21
3.3.5 Interrupt Level Mask Register (ILM) 22
3.3.6 Condition Code Register (CCR) 23
3.3.7 System Condition Code Register (SCR) 25
3.3.8 Return Pointer (RP) 26
3.3.9 System Stack Pointer (SSP) 27
3.3.10 User Stack Pointer (USP) 28
3.3.11 Table Base Register (TBR) 29
3.3.12 Multiplication/Division Register (MDH, MDL) 30
3.3.13 Base Pointer (BP) 32
3.3.14 FPU Control Register (FCR) 32
3.3.15 Exception status register (ESR) 37
3.3.16 Debug Register (DBR) 39
3.4 Floating-point Register 40

CHAPTER 4 RESET AND "EIT" PROCESSING 41
4.1 Reset 42
4.2 Basic Operations in EIT Processing 43
4.2.1 Types of EIT Processing and Prior Preparation 43
4.2.2 EIT Processing Sequence 44
4.2.3 Recovery from EIT Processing 45
4.3 Processor Operation Status 46
4.4 Exception Processing 48
4.4.1 Invalid Instruction Exception 48
4.4.2 Instruction Access Protection Violation Exception 49
4.4.3 Data Access Protection Violation Exception 49
4.4.4 FPU Exception 50
4.4.5 Instruction Break 51
4.4.6 Guarded Access Break 52
4.5 Interrupts 53
4.5.1 General interrupts 53
4.5.2 Non-maskable Interrupts (NMI) 55
4.5.3 Break Interrupt 55
4.5.4 Data Access Error Interrupt 56
4.6 Traps 57
4.6.1 INT Instructions 57
4.6.2 INTE Instruction 57
4.6.3 Step Trace Traps 58
4.7 Multiple EIT processing and Priority Levels 60
4.7.1 Multiple EIT Processing 60
4.7.2 Priority Levels of EIT Requests 61
4.7.3 EIT Acceptance when Branching Instruction is Executed 62
4.8 Timing When Register Settings Are Reflected 63
4.8.1 Timing when the interrupt enable flag (I) is requested 63
4.8.2 Timing of Reflection of Interrupt Level Mask Register (ILM) 64
4.9 Usage Sequence of General Interrupts 65
4.9.1 Preparation while using general interrupts 65
4.9.2 Processing during an Interrupt Processing Routine 66
4.9.3 Points of Caution while using General Interrupts 66
4.10 Precautions 67
4.10.1 Exceptions in EIT Sequence and RETI Sequence 67
4.10.2 Exceptions in Multiple Load and Multiple Store Instructions 67
4.10.3 Exceptions in Direct Address Transfer Instruction 67

CHAPTER 5 PIPELINE OPERATION 69
5.1 Instruction execution based on Pipeline 70
5.1.1 Integer Pipeline 70
5.1.2 Floating Point Pipeline 72
5.2 Pipeline Operation and Interrupt Processing 73
5.2.1 Mismatch in Acceptance and Cancellation of Interrupt 73
5.2.2 Method of preventing the mismatched pipeline conditions 73
5.3 Pipeline hazards 74

5.3.1 Occurrence of data hazard 74
5.3.2 Register Bypassing 74
5.3.3 Interlocking 75
5.3.4 Interlocking produced by reference to R15 after Changing the Stack flag (S) 75
5.3.5 Structural Hazard 75
5.3.6 Control Hazard 76
5.4 Non-block loading 77
5.5 Delayed branching processing 78
5.5.1 Example of branching with non-delayed branching instructions 78
5.5.2 Example of processing of delayed branching instruction 79

CHAPTER 6 INSTRUCTION OVERVIEW 81
6.1 Instruction System 82
6.1.1 Integer Type Instructions 82
6.1.2 Floating Point Type Instructions 84
6.2 Instructions Formats 85
6.2.1 Instructions Notation Formats 85
6.2.2 Addressing Formats 86
6.2.3 Instruction Formats 87
6.2.4 Register designated Field 91
6.3 Data Format 93
6.3.1 Data Format Used by Integer Type Instructions (Common with All FR Family) 93
6.3.2 Format Used for Floating Point Type Instructions 94
6.4 Read-Modify-Write type Instructions 96
6.5 Branching Instructions and Delay Slot 97
6.5.1 Delayed Branching Instructions 97
6.5.2 Specific example of Delayed Branching Instructions 98
6.5.3 Non-Delayed Branching Instructions 99
6.6 Step Division Instructions 100
6.6.1 Signed Division 100
6.6.2 Unsigned Division 101

CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS 103
7.1 ADD (Add 4bit Immediate Data to Destination Register) 105
7.2 ADD (Add Word Data of Source Register to Destination Register) 107
7.3 ADD2 (Add 4bit Immediate Data to Destination Register) 109
7.4 ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) 111
7.5 ADDN (Add Immediate Data to Destination Register) 113
7.6 ADDN (Add Word Data of Source Register to Destination Register) 115
7.7 ADDN2 (Add Immediate Data to Destination Register) 117
7.8 ADDSP (Add Stack Pointer and Immediate Data) 119
7.9 AND (And Word Data of Source Register to Data in Memory) 121
7.10 AND (And Word Data of Source Register to Destination Register) 123
7.11 ANDB (And Byte Data of Source Register to Data in Memory) 125
7.12 ANDCCR (And Condition Code Register and Immediate Data) 127
7.13 ANDH (And Halfword Data of Source Register to Data in Memory) 129
7.14 ASR (Arithmetic shift to the Right Direction) 131

7.15 ASR (Arithmetic shift to the Right Direction) 133
7.16 ASR2 (Arithmetic shift to the Right Direction) 135
7.17 BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory) 137
7.18 BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory) 139
7.19 Bcc (Branch relative if Condition satisfied) 141
7.20 Bcc:D (Branch relative if Condition satisfied) 143
7.21 BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory) 145
7.22 BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory) 147
7.23 BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory) 149
7.24 BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory) 151
7.25 BTSTH (Test Higher 4bit of Byte Data in Memory) 153
7.26 BTSTL (Test Lower 4bit of Byte Data in Memory) 155
7.27 CALL (Call Subroutine) 157
7.28 CALL (Call Subroutine) 159
7.29 CALL:D (Call Subroutine) 161
7.30 CALL:D (Call Subroutine) 163
7.31 CMP (Compare Immediate Data and Destination Register) 165
7.32 CMP (Compare Word Data in Source Register and Destination Register) 167
7.33 CMP2 (Compare Immediate Data and Destination Register) 169
7.34 DIV0S (Initial Setting Up for Signed Division) 171
7.35 DIV0U (Initial Setting Up for Unsigned Division) 173
7.36 DIV1 (Main Process of Division) 175
7.37 DIV2 (Correction When Remain is zero) 177
7.38 DIV3 (Correction When Remain is zero) 179
7.39 DIV4S (Correction Answer for Signed Division) 181
7.40 DMOV (Move Word Data from Direct Address to Register) 183
7.41 DMOV (Move Word Data from Register to Direct Address) 185
7.42 DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) 187
7.43 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 189
7.44 DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) 191
7.45 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 193
7.46 DMOVB (Move Byte Data from Direct Address to Register) 195
7.47 DMOVB (Move Byte Data from Register to Direct Address) 197
7.48 DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) 199
7.49 DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) 201
7.50 DMOVH (Move Halfword Data from Direct Address to Register) 203
7.51 DMOVH (Move Halfword Data from Register to Direct Address) 205
7.52 DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) 207
7.53 DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) 209
7.54 ENTER (Enter Function) 211
7.55 EOR (Exclusive Or Word Data of Source Register to Data in Memory) 213
7.56 EOR (Exclusive Or Word Data of Source Register to Destination Register) 215
7.57 EORB (Exclusive Or Byte Data of Source Register to Data in Memory) 217
7.58 EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) 219
7.59 EXTSB (Sign Extend from Byte Data to Word Data) 221
7.60 EXTSH (Sign Extend from Byte Data to Word Data) 223

7.61 EXTUB (Unsign Extend from Byte Data to Word Data) 225
7.62 EXTUH (Unsign Extend from Byte Data to Word Data) 227
7.63 FABSs (Single Precision Floating Point Absolute Value) 229
7.64 FADDs (Single Precision Floating Point Add) 230
7.65 FBcc (Floating Point Conditional Branch) 232
7.66 FBcc:D (Floating Point Conditional Branch with Delay Slot) 234
7.67 FCMPs (Single Precision Floating Point Compare) 236
7.68 FDIVs (Single Precision Floating Point Division) 238
7.69 FiTOs (Convert from Integer to Single Precision Floating Point) 240
7.70 FLD (Single Precision Floating Point Data Load) 242
7.71 FLD (Single Precision Floating Point Data Load) 243
7.72 FLD (Single Precision Floating Point Data Load) 244
7.73 FLD (Single Precision Floating Point Data Load) 245
7.74 FLD (Single Precision Floating Point Data Load) 246
7.75 FLD (Load Word Data in Memory to Floating Register) 247
7.76 FLDM (Single Precision Floating Point Data Load to Multiple Register) 248
7.77 FMADDs (Single Precision Floating Point Multiply and Add) 250
7.78 FMOVs (Single Precision Floating Point Move) 252
7.79 FMSUBs (Single Precision Floating Point Multiply and Subtract) 253
7.80 FMULs (Single Precision Floating Point Multiply) 255
7.81 FNEGs (Single Precision Floating Point sign reverse) 257
7.82 FSQRTs (Single Precision Floating Point Square Root) 258
7.83 FST (Single Precision Floating Point Data Store) 259
7.84 FST (Single Precision Floating Point Data Store) 260
7.85 FST (Single Precision Floating Point Data Store) 261
7.86 FST (Single Precision Floating Point Data Store) 262
7.87 FST (Single Precision Floating Point Data Store) 263
7.88 FST (Store Word Data in Floating Point Register to Memory) 264
7.89 FSTM (Single Precision Floating Point Data Store from Multiple Register) 265
7.90 FsTOi (Convert from Single Precision Floating Point to Integer) 267
7.91 FSUBs (Single Precision Floating Point Subtract) 269
7.92 INT (Software Interrupt) 271
7.93 INTE (Software Interrupt for Emulator) 273
7.94 JMP (Jump) 275
7.95 JMP:D (Jump) 277
7.96 LCALL (Long Call Subroutine) 279
7.97 LCALL:D (Long Call Subroutine) 280
7.98 LD (Load Word Data in Memory to Register) 281
7.99 LD (Load Word Data in Memory to Register) 283
7.100 LD (Load Word Data in Memory to Register) 285
7.101 LD (Load Word Data in Memory to Register) 287
7.102 LD (Load Word Data in Memory to Register) 289
7.103 LD (Load Word Data in Memory to Register) 291
7.104 LD (Load Word Data in Memory to Register) 292
7.105 LD (Load Word Data in Memory to Program Status Register) 294
7.106 LDI:20 (Load Immediate 20bit Data to Destination Register) 296
7.107 LDI:32 (Load Immediate 32 bit Data to Destination Register) 298

7.108 LDI:8 (Load Immediate 8bit Data to Destination Register) 300
7.109 LDM0 (Load Multiple Registers) 302
7.110 LDM1 (Load Multiple Registers) 304
7.111 LDUB (Load Byte Data in Memory to Register) 306
7.112 LDUB (Load Byte Data in Memory to Register) 308
7.113 LDUB (Load Byte Data in Memory to Register) 310
7.114 LDUB (Load Byte Data in Memory to Register) 312
7.115 LDUH (Load Halfword Data in Memory to Register) 313
7.116 LDUH (Load Halfword Data in Memory to Register) 315
7.117 LDUH (Load Halfword Data in Memory to Register) 317
7.118 LDUH (Load Halfword Data in Memory to Register) 319
7.119 LEAVE (Leave Function) 320
7.120 LSL (Logical Shift to the Left Direction) 322
7.121 LSL (Logical Shift to the Left Direction) 324
7.122 LSL2 (Logical Shift to the Left Direction) 326
7.123 LSR (Logical Shift to the Right Direction) 328
7.124 LSR (Logical Shift to the Right Direction) 330
7.125 LSR2 (Logical Shift to the Right Direction) 332
7.126 MOV (Move Word Data in Source Register to Destination Register) 334
7.127 MOV (Move Word Data in Source Register to Destination Register) 336
7.128 MOV (Move Word Data in Program Status Register to Destination Register) 338
7.129 MOV (Move Word Data in Source Register to Destination Register) 340
7.130 MOV (Move Word Data in Source Register to Program Status Register) 342
7.131 MOV (Move Word Data in General Purpose Register to Floating Point Register) 344
7.132 MOV (Move Word Data in Floating Point Register to General Purpose Register) 345
7.133 MUL (Multiply Word Data) 346
7.134 MULH (Multiply Halfword Data) 348
7.135 MULU (Multiply Unsigned Word Data) 350
7.136 MULUH (Multiply Unsigned Halfword Data) 352
7.137 NOP (No Operation) 354
7.138 OR (Or Word Data of Source Register to Data in Memory) 356
7.139 OR (Or Word Data of Source Register to Destination Register) 358
7.140 ORB (Or Byte Data of Source Register to Data in Memory) 360
7.141 ORCCR (Or Condition Code Register and Immediate Data) 362
7.142 ORH (Or Halfword Data of Source Register to Data in Memory) 364
7.143 RET (Return from Subroutine) 366
7.144 RET:D (Return from Subroutine) 368
7.145 RETI (Return from Interrupt) 370
7.146 SRCH0 (Search First Zero bit position distance From MSB) 373
7.147 SRCH1 (Search First One bit position distance From MSB) 375
7.148 SRCHC (Search First bit value change position distance From MSB) 377
7.149 ST (Store Word Data in Register to Memory) 379
7.150 ST (Store Word Data in Register to Memory) 381
7.151 ST (Store Word Data in Register to Memory) 383
7.152 ST (Store Word Data in Register to Memory) 385
7.153 ST (Store Word Data in Register to Memory) 387
7.154 ST (Store Word Data in Register to Memory) 389

7.155 ST (Store Word Data in Register to Memory) ... 390
7.156 ST (Store Word Data in Program Status Register to Memory) ... 392
7.157 STB (Store Byte Data in Register to Memory) ... 394
7.158 STB (Store Byte Data in Register to Memory) ... 396
7.159 STB (Store Byte Data in Register to Memory) ... 398
7.160 STB (Store Byte Data in Register to Memory) ... 400
7.161 STH (Store Halfword Data in Register to Memory) ... 401
7.162 STH (Store Halfword Data in Register to Memory) ... 403
7.163 STH (Store Halfword Data in Register to Memory) ... 405
7.164 STH (Store Halfword Data in Register to Memory) ... 407
7.165 STILM (Set Immediate Data to Interrupt Level Mask Register) ... 408
7.166 STM0 (Store Multiple Registers) ... 410
7.167 STM1 (Store Multiple Registers) ... 412
7.168 SUB (Subtract Word Data in Source Register from Destination Register) ... 414
7.169 SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) ... 416
7.170 SUBN (Subtract Word Data in Source Register from Destination Register) ... 418
7.171 XCHB (Exchange Byte Data) ... 420

APPENDIX ... 423
APPENDIX A Instruction Lists ... 424
A.1 Meaning of Symbols ... 425
A.1.1 Mnemonic and Operation Columns ... 425
A.1.2 Operation Column ... 430
A.1.3 Format Column ... 431
A.1.4 OP Column ... 431
A.1.5 CYC Column ... 432
A.1.6 FLAG Column ... 433
A.1.7 RMW Column ... 433
A.1.8 Reference Column ... 433
A.2 Instruction Lists ... 434
A.3 List of Instructions that can be positioned in the Delay Slot ... 448
APPENDIX B Instruction Maps ... 450
B.1 Instruction Maps ... 451
B.2 Extension Instruction Maps ... 452
APPENDIX C Supplemental Explanation about FPU Exception Processing ... 455
C.1 Conformity with IEEE754-1985 Standard ... 455
C.2 FPU Exceptions ... 456
C.3 Round Processing ... 458

INDEX ... 461

CHAPTER 1

OVERVIEW OF FR81 FAMILY CPU

This chapter describes the features of FR81 Family CPU and the changes from the earlier FR Family.

1.1 Features of FR81 Family CPU

1.2 Changes from the earlier FR Family

1.1 Features of FR81 Family CPU

FR81 Family CPU is meant for 32 bit RISC controller having proprietary FR81 architecture of Fujitsu. The FR81 architecture is optimized for microcontrollers by using the FR family instruction set and including improved floating-point, memory protection, and debug functions.

■ General-purpose Register Architecture

It is load/store architecture based on 16 numbers of 32-bit General-purpose registers R0 to R15. The architecture also has instructions that are suitable for embedded uses such as memory to memory transfer, bit processing etc.

■ Linear Space for 32-bit (4G bytes) addressing

Address space is controlled for each byte unit. Linear specification of Address is made based on 32-bit address.

■ 16-bit fixed instruction length (excluding immediate data transfer instructions)

It is 16-bit fixed length instruction format excluding 32/20-bit immediate data transfer instruction. It enables securing high object efficiency.

■ Floating point calculation unit (FPU)

FR81 Family supports single precision floating point calculation (IEEE754 compliant). It has 16 pieces of 32-bit floating point registers from FR0 to FR15. A single instruction can execute a product-sum operation type calculation (multiplication, or addition/subtraction). The instruction length of a floating point type instruction is 32 bits

■ Pipeline Configuration

High speed one-instruction one-cycle processing of the basic instructions based on 5-stage pipeline operation can be carried out. Pipeline has following 5-stage configuration.

- IF Stage: Load Instruction
- ID Stage: Interpret Instruction
- EX Stage: Execute Instruction
- MA Stage: Memory Access
- WB Stage: Write to register

FR81 Family has the 6-stage pipeline configuration to execute floating point type instructions.

■ Non-blocking load

In FR81 Family, non-blocking loading is carried out making execution of LD (load) instructions efficient. A maximum of four LD (Load) instructions can be issued in anticipation. In non-blocking, succeeding instruction is executed without waiting for the completion of a load instruction, in case general-purpose register storing the value of load instruction is not referred by the succeeding instruction.

FR81 Family

■ Harvard Architecture

An instruction can be executed efficiently based on Harvard Architecture where instruction bus for instruction access and data bus for data access are independent.

■ Multiplication Instruction

Multiplication/division computation can be executed at the instruction level based on an in-built multiplier. 32-bit multiplication, signed or unsigned, is executed in 5 cycles. 16-bit multiplication is executed in 3 cycles.

■ Step Division Instruction

32-bit ÷ 32-bit division, signed or unsigned, can be executed based on combination of step division instructions.

■ Direct Addressing Instruction for peripheral access

Address of 256 words/ 256 half-words/ 256 bytes from the top of address space (low order address) can be directly specified. It is convenient for address specification in the I/O Register of the peripheral resource.

■ High-speed interrupt processing complete within 6 cycles

Acceptance of interruption is processed at a high speed within 6 cycles. A 16-level priority order is given to the request for interruption. Masking in line with the priority order can be carried out based on interruption mask level of the CPU.

1.2 Changes from the earlier FR Family

FR81 Family has partial addition and deletion of instructions and operational changes from the earlier FR Family (FR30 Family, FR60 Family etc.).

■ Instructions that cannot be used in FR81/FR80 Family

Following instructions cannot be used in FR81/FR80 Family.

- Coprocessor Instructions (COPOP, COPLD, COPST, COPSV)
- Resource Instructions (LDRES, STRES)

Undefined Instruction Exceptions and not the Coprocessor Error Trap occur when execution of Coprocessor Instruction is attempted. Undefined Instruction Exceptions occur when execution of Resource Instruction is attempted.

■ Instructions added to FR81/FR80 Family

Following instructions have been added in FR81/FR80 Family. These instructions have replaced the bit search module embedded as a peripheral function.

- SRCH1 (Bit Search Instruction Detection of First "1" bit from MSB to LSB)
- SRCH0 (Bit Search Instruction Detection of first "0" bit from MSB to LSB)
- SRCHC (Bit Search Instruction Detection of Change point from MSB to LSB)

see "Chapter 7 Detailed Execution Instructions" and "Appendix A 2 Instruction Lists" for operation of Bit Search Instructions.

■ Adding floating point type instructions

Floating point type instructions and 16 pieces of 32-bit floating point registers (FR0 to FR15) have been added in FR81 Family.

■ Privilege mode

Privilege mode has been added in FR81 family. Privilege mode and user mode are two CPU operation modes.

■ Exception processing

Exception processing has been improved for FR81 Family. The following exceptions have been added.

- FPU exception
- Instruction access protection violation exception
- Data access protection violation exception
- Invalid instruction exception (Changing definition from undefined instruction exception)
- Data access error exception
- FPU absence exception

FR81 Family

■ Operation of INTE Instructions during Step Execution

In FR81 Family, trap processing is initiated based on INTE instructions even during step execution based on step trace trap.

In hitherto FR Family, trap processing is not initiated based on INTE instructions during step execution.

For trap processing based on step trace trap and INTE instructions, see "4.6 Traps".

CHAPTER 2

MEMORY ARCHITECTURE

This chapter explains the memory architecture of FR81 Family CPU. Memory architecture refers to allocation of memory spaces and methods used to access memory.

2.1 Address Space

2.2 Data Structure

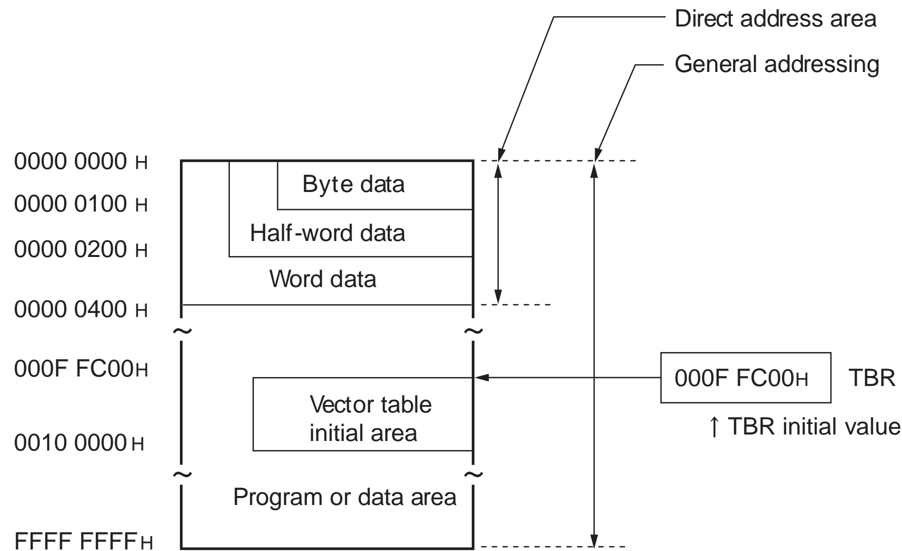
2.3 Word Alignment

2.1 Address Space

The address space of FR81 Family CPU is 32 bits (4Gbyte).

CPU controls the address spaces in byte units. An address on the address space is accessed from the CPU by specifying a 32-bit value. Address space is indicated in Figure 2.1-1.

Figure 2.1-1 Address space



Address space is also called memory space. It is a logical address space as seen from the CPU. Addresses cannot be changed. Logical address as seen from the CPU, and the physical address actually allocated to memory or I/O are identical.

2.1.1 Direct Address Area

In the lower address in the address space, there is a direct address area.

Direct address area directly specifies an address in the direct address specification instruction. This area accesses only based on operand data in the instruction without the use of general-purpose registers. The size of the address area that can be specified by direct addressing varies according to the data type being accessed.

The correspondence between data type and area specified by direct address is as follows.

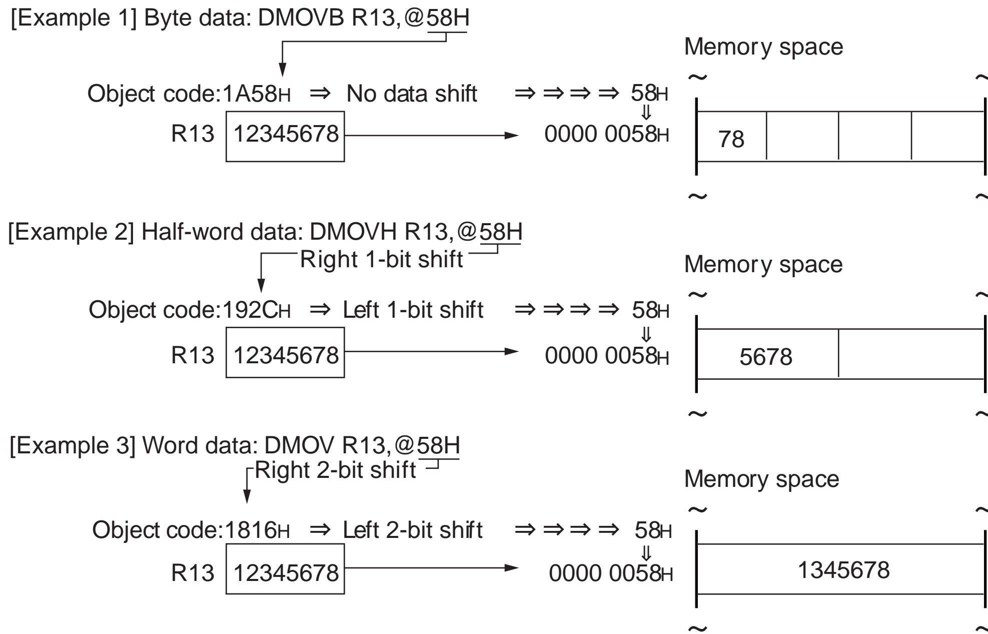
- byte data access: 0000 0000_H to 0000 $00FF_H$
- half-word data access: 0000 0000_H to 0000 $01FF_H$
- word data access: 0000 0000_H to 0000 $03FF_H$

The method of using the 8-bit address data contained in the operand of instructions that specify direct addresses is as follows:

- byte data access: Lower 8 bits of the address are used as it is
- half word data access: Value is doubled and used as lower 9 bits of the address
- word data access: Value is quadrupled and used as lower 10 bits of the address

The relation between data types specified by direct address and memory address is shown in Figure 2.1-2.

Figure 2.1-2 Relation between data type specified by direct address and memory address

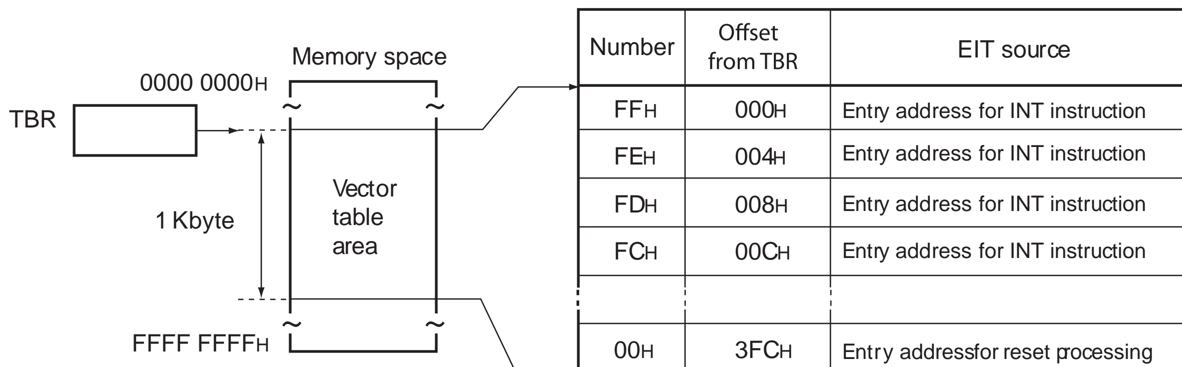


2.1.2 Vector Table Area

An area of 1Kbyte from the address shown in the Table Base Register (TBR) is called the EIT Vector Table Area.

Table Base Register (TBR) represents the top address of the vector table area. In this vector table area, the entry addresses of EIT processing (Exception processing, Interrupt processing, Trap processing) are described. The relation between Table Base Register (TBR) and vector table area is shown in Figure 2.1-3.

Figure 2.1-3 Relation between Table Base Register (TBR) and Vector Table Area addresses



Number	Offset from TBR	EIT source
FFH	000H	Entry address for INT instruction
FEH	004H	Entry address for INT instruction
FDH	008H	Entry address for INT instruction
FCH	00CH	Entry address for INT instruction
00H	3FCH	Entry addressfor reset processing

As a result of reset, the value of Table Base Register (TBR) is initialized to 000F $FC00_H$, and the range of vector table area extends from 000F $FC00_H$ to 000F $FFFF_H$. By rewriting the Table Base Register (TBR), the vector table area can be allocated to any desired location.

A vector table is composed of entry addresses for each EIT processing programs. Each vector table contains values whose use is fixed according to the CPU architecture, and values that vary according to the type of built-in peripheral functions. The structure of vector table area is shown in Table 2.1-1.

Table 2.1-1 Structure of Vector Table Area

Offset from TBR	Vector number	Model-dependence	EIT value description	Remarks
$3FC_H$	00_H	No	reset	
$3F8_H$	01_H	No	system reserved	
$3F4_H$	02_H	No	system reserved	Disabled
$3F0_H$	03_H	No	system reserved	Disabled
$3EC_H$	04_H	No	system reserved	Disabled
$3E8_H$	05_H	No	FPU exception	
$3E4_H$	06_H	No	Instruction access protection violation exception	
$3E0_H$	07_H	No	Data access protection violation exception	
$3DC_H$	08_H	No	Data access error interrupt	
$3D8_H$	09_H	No	INTE instruction	For use in the emulator
$3D4_H$	$0A_H$	No	Instruction break	
$3D0_H$	$0B_H$	No	system reserved	
$3CC_H$	$0C_H$	No	Step trace trap	
$3C8_H$	$0D_H$	No	system reserved	
$3C4_H$	$0E_H$	No	Invalid instruction exception	
$3C0_H$	$0F_H$	No	NMI request	
$3BC_H$ to 304_H	$0F_H$ to $3E_H$	Yes	General interrupt (used in external interrupt, interrupt from peripheral function)	Refer to the Hardware Manual for each model
300_H	$3F_H$	No	General interrupts	Used in Delayed interrupt
$2FC_H$	40_H	No	system reserved	Used in REALOS
$2F8_H$	41_H	No	system reserved	Used in REALOS
$2F4_H$ to 000_H	42_H to FF_H	No	Used in INT instruction	

For vector tables of actual models, refer to the hardware manuals for each model.

2.1.3 20-bit Addressing Area & 32-bit Addressing Area

The lower portion of the address space extending from 0000 0000_H to 000F $FFFF_H$ (1Mbyte) will be the 20-bit addressing area. The overall address space from 0000 0000_H to FFFF $FFFF_H$ will be 32-bit addressing space.

If all the program locations and data locations are positioned within the 20-bit addressing area, a compact and high-speed program can be realized as compared to a 32-bit addressing area.

In a 20-bit addressing area, as the address values are within 20 bits, the LDI:20 instruction can be used for immediate loading of address information. The instruction length (Code size) of LDI:20 instruction is 4bytes. By using LDI:20 instruction, the program becomes more compact than when using LDI:32 instruction of instruction length 6bytes.

Example of 20-bit Addressing

```
                            Code size
LDI:20    #label20,Ri      ; 4 bytes
JMP       @Ri              ; 2 bytes
                            Total 6 bytes
```

Example of 32-bit Addressing

```
                            Code size
LDI:32    #label32,Ri      ; 6 bytes
JMP       @Ri              ; 2 bytes
                            Total 8 bytes
```

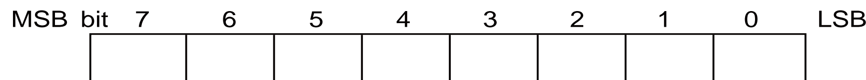
2.2 Data Structure

FR81 Family CPU has three data types namely byte data (8-bits), half word data (16-bits) and word data (32-bits). The byte order is big endian.

2.2.1 Byte Data

This is a data type having 8 bits as unit. Bit order is little endian, MSB side becomes bit7 and LSB side becomes bit0. The structure of byte data is shown in Figure 2.2-1.

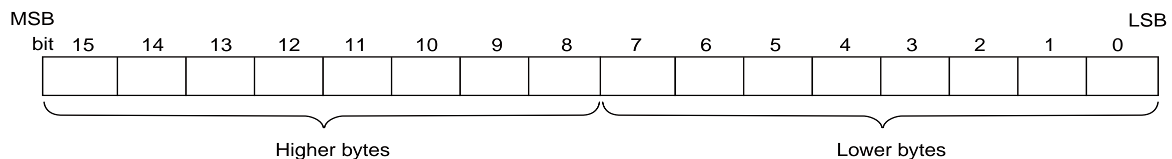
Figure 2.2-1 Structure of byte data



2.2.2 Half Word Data

This is a data type having 16 bits (2byte) as unit. Bit order is little endian, MSB side is bit15 while LSB side is bit0. Bit15 to bit8 of MSB side represent the higher bytes while bit7 to bit0 of LSB side represent the lower bytes. The structure of half word data is shown in Figure 2.2-2.

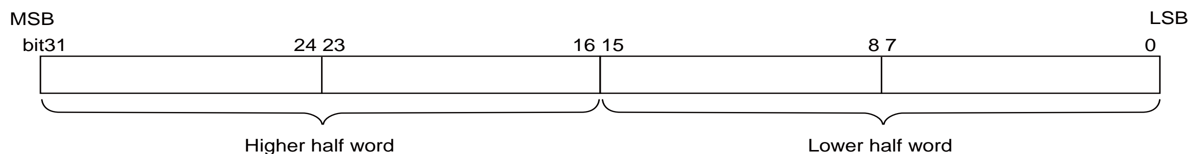
Figure 2.2-2 Structure of Half Word Data



2.2.3 Word Data

This is a data type having 32 bits (4byte) as unit. Bit order is little endian, MSB side is bit31 while LSB side is bit0. Bit31 to bit16 of the MSB side become the higher half word, while bit15 to bit0 of the LSB side become the lower half word. The structure of word data is shown in Figure 2.2-3.

Figure 2.2-3 Structure of Word Data



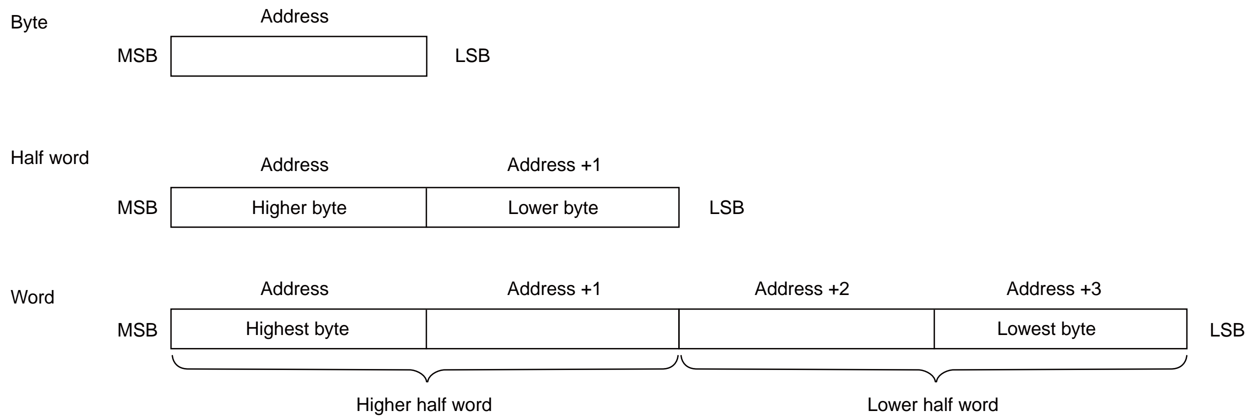
FR81 Family

2.2.4 Byte Order

The byte order of FR81 Family CPU is big endian. When word data or half word data are allocated to address spaces, the higher bytes are placed in the lower address side while the lower bytes are placed in the higher address side. The arrangement of big endian byte data is shown in Figure 2.2-4.

For example, if a word data was written on the memory (RAM) at address location 0004 1234_H of the memory space, the highest byte will be stored at location 0004 1234_H while the lowest byte will be stored at location byte 0004 1237_H.

Figure 2.2-4 Big Endian Byte Orde



2.3 Word Alignment

The data type used determines restrictions on the designation of memory addresses (word alignment).

2.3.1 Program Access

Unit of instruction length is half word (2byte) and all instructions are allocated to addresses which are multiples of 2 (2n location).

At the time of execution of the instruction, bit0 of the program counter (PC) automatically becomes "0", and is always at an even address. In a branched instruction, even if an odd address is generated as a result of branch destination address calculation, the bit0 of the address will be assigned "0" and branched to an even address.

There is no address exception in program access.

2.3.2 Data Access

There are following restrictions on addresses for data access depending upon the data type used.

Word data

Data is assigned to addresses that are multiples of 4 (4n location). The restriction of multiples of 4 on addresses is called ‘word boundary’. If the specified address is not a multiple of 4, the lower two bits of the address are set to "00" forcibly.

Half-word data

Data is assigned to addresses that are multiples of 2 (2n locations). The restriction of multiples of 2 on addresses is called ‘half-word boundary’. If the specified address is not a multiple of 2, the lower 1 bit of the address is set to "0" forcibly.

Byte data

There is no restriction on allocation of addresses.

During word and half-word data access, condition that lower bit of an address has to be "0" is applicable only for the result of computation of an effective address. Values still under calculation are used as they are.

CHAPTER 3
PROGRAMMING MODEL

This chapter describes the programming model of FR81 Family CPU.

3.1 Register Configuration
3.2 General-purpose Registers
3.3 Dedicated Registers
3.4 Floating-point Register

3.1 Register Configuration

FR81 Family CPU uses three types of registers, namely, general-purpose registers, dedicated registers and floating point registers.

General-purpose registers are registers that store computation data and address information. They comprise 16 registers from R0 to R15. Dedicated registers are registers that store information for specific applications.

Floating point registers are registers that store calculation information for floating point calculations. They are comprised of 16 registers from FR0 to FR15.

FR81 Family

3.2 General-purpose Registers

General-purpose registers are used for storing results of various calculations, as well as information about addresses to be used as pointers for memory access.

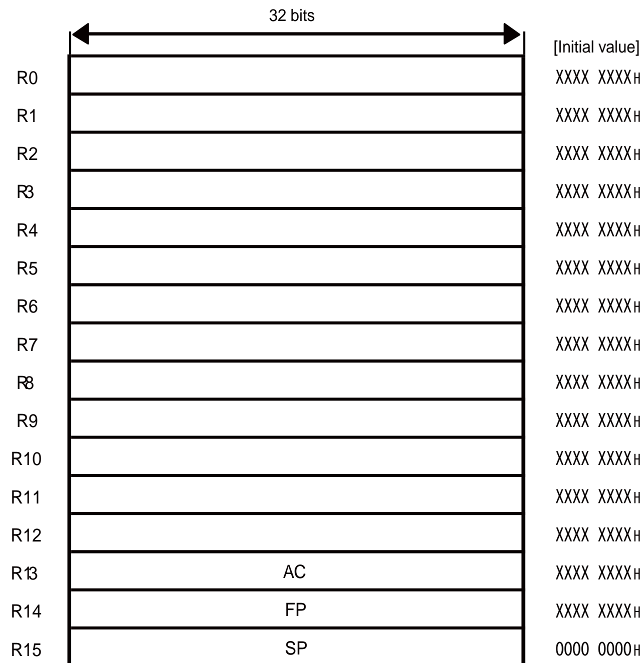
3.2.1 Configuration of General-purpose Registers

General-purpose registers has sixteen each 32 bits in length. General-purpose registers have names R0 to R15.

In case of general instructions, the general-purpose registers can use without any distinction. In some instructions, three registers namely R13, R14 and R15 have special usages.

Figure 3.2-1 shows the configuration and initial values of general-purpose registers.

Figure 3.2-1 Configuration and initial values of general-purpose registers



R0 to R14 are not initialized as a result of reset. R15 is initialized 0000 0000_H as a result of reset.

3.2.2 Special Usage of General-purpose Registers

General-purpose registers R13 to R15, besides being used as other general-purpose registers, are used in the following way in some instructions.

R13 (Virtual Accumulator: AC)

- Base address register for load/store to memory instructions
  [Example: LD @(R13,Rj), Ri]
- Accumulator for direct address designation
  [Example: DMOV @dir10, R13]
- Memory pointer for direct address designation
  [Example: DMOV @dir10,@R13+]

R14 (Frame Pointer: FP)

- Index register for load/store to memory instructions
  [Example: LD @(R14,disp10), Ri]
- Frame pointer for reserve/release of dynamic memory area
  [Example: ENTER #u10]

R15 (Stack Pointer: SP)

- Index register for load/store to memory instructions
  [Example: LD @(R15,udisp6), Ri]
- Stack pointer
  [Example: LD @R15+,Ri]
- Stack pointer for reserve/release of dynamic memory area
  [Example: ENTER #u10]

3.2.3 Relation between Stack Pointer and R15

R15 functions as an indirect register. Physically it becomes either the system stack pointer (SSP) or user pointer (USP) for dedicated registers. When the notation R15 is used in an instruction, this register will function as USP if the stack flag (S) is "1" and as SSP if the stack flag is "0". Table 3.2-1 shows the correlation between general-purpose register R15 and stack pointer.

When something is written on R15 as a general-purpose register, it is automatically written onto the system stack pointer (SSP) or user stack pointer (USP) according to the value of stack flag (S).

Table 3.2-1 Correlation between General-purpose Register "R15" and Stack Pointer

General-purpose register	S Flag	Stack pointer
R15	1	User stack pointer (USP)
	0	System stack pointer (SSP)

Stack flag (S) is present in the condition code register (CCR) section of the program status (PS).

FR81 Family

3.3 Dedicated Registers

FR81 Family CPU has dedicated registers reserved for special usages.

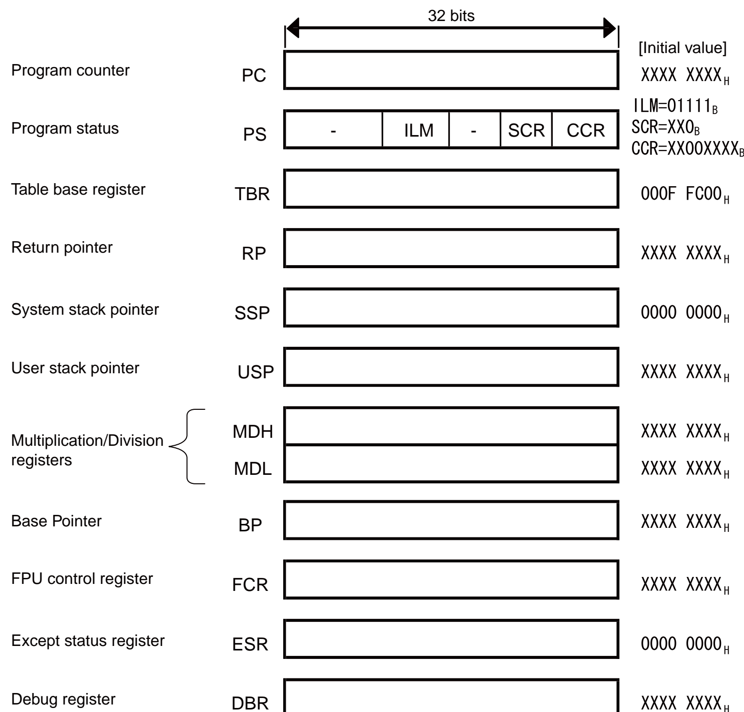
3.3.1 Configuration of Dedicated Registers

Dedicated registers are used for special purposes. The following dedicated registers are available.

- Program counter (PC)
- Program status (PS)
- Return pointer (RP)
- System stack pointer (SSP)
- User stack pointer (USP)
- Table base register (TBR)
- Multiplication/Division Register (MDH, MDL)
- Base Pointer (BP)
- FPU control register (FCR)
- Exception status register (ESR)
- Debug register (DBR)

Figure 3.3-1 shows the configuration and initial values of dedicated registers.

Figure 3.3-1 Configuration and Initial Values of Dedicated Registers



3.3.2 Program Counter (PC)

Program counter (PC) is a 32-bit register that indicates the address containing the instruction that is currently executing.

Figure 3.3-2 shows the bit configuration of program counter (PC).

Figure 3.3-2 Program Counter (PC) Bit Configuration

The value of the lowest bit (LBS) of the program counter (PC) is always read as “0”. Even if "1" is written to it as a result of address calculation of branching destination, the lowest bit of branching address will be treated as "0". When the program counter (PC) changes after the execution of an instruction and it indicates the next instruction, the lowest bit is always read as "0".

Following a reset, the contents of the Program Counter (PC) are set to the value (reset entry address) written in the reset vector of the vector table. As the table base register (TBR) is initialized first by reset, the address of the reset vector will be 000F $FFFC_H$.

3.3.3 Program Status (PS)

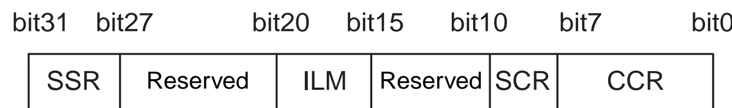
Program status (PS) is a 32-bit register that indicates the status of program execution. It sets the interrupt enable level, controls the program trace break function in the CPU, and indicates the status of instruction execution.

Program status (PS) consists of the following 4 parts.

- System status register (SSR)
- Interrupt level mask register (ILM)
- System condition code register (SCR)
- Condition code register (CCR)

Figure 3.3-3 shows the bit configuration of program status (PS).

Figure 3.3-3 Program status (PS) Bit Configuration



The reserved bits of program status (PS) are all reserved for future expansion. The read value of reserved bits is always "0". Write values should always be written as "0".

FR81 Family

3.3.4 System Status Register (SSR)

System status register (SSR) is a 4-bit register that indicates the state of the CPU. It lies between bit 31 and bit 28 of the program status (PS).

Figure 3.3-4 shows the bit configuration of system status register (SSR).

Figure 3.3-4 System Status Register (SSR) Bit Configuration

bit31	bit30	bit29	bit28	Initial value
DBG	UM	FPU	MPU	0011_B

The contents of each bit are described below.

[bit31] DBG: Debug State Flag

This flag indicates the debugging state during debugging. The flag bit is turned to "1" when the system shifts to a debug state, and turned to "0" when moving from the debug state with a RETI instruction. This cannot be rewritten using instructions such as the MOV instruction.

The initial value of the debug state flag (DBG) after a reset is "0".

[bit30] UM: User Mode Flag

This flag indicates the user mode. The flag bit is turned to "1" when the system is shifted to user mode by the execution of a RETI instruction, and cleared to "0" when shifted to privilege mode with EIT. Upon execution of the RETI instruction, if bit 30 of the PS value is set to "1", a value returned from memory, the system shifts to user mode. This cannot be rewritten using instructions such as the MOV instruction.

The initial value of the user mode flag (UM) after a reset is "0".

[bit29] FPU: FPU presence flag

This flag indicates that the floating point calculation unit (FPU) is installed. The flag bit is set to "1" if a FPU is installed, and "0" if it is not the case. This bit cannot be rewritten.

Table 3.3-1 FPU presence flag (FPU) in the system status register

flag	value	Meaning
FPU	0	With FPU (installed)
	1	Without FPU (not installed)

[bit28] MPU: MPU presence flag

This flag indicates that the memory protection unit (MPU) is installed. The flag bit is set to "1" if a MPU is installed, and "0" if it is not the case. This bit cannot be rewritten.

Table 3.3-2 MPU presence flag (MPU) in the system status register

flag	value	Meaning
MPU	0	With MPU (installed)
	1	Without MPU (not installed)

3.3.5 Interrupt Level Mask Register (ILM)

Interrupt level mask register (ILM) is a 5-bit register used to store the interrupt level mask value. It lies between bit20 to bit16 of the program status (PS).

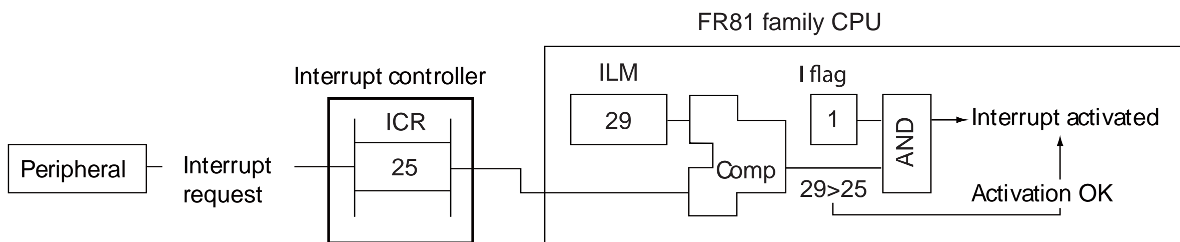
Figure 3.3-5 shows the bit configuration of interrupt level mask register (ILM).

Figure 3.3-5 Interrupt Level Mask Register (ILM) Bit Configuration

bit20	bit19	bit18	bit17	bit16	Initial value
ILM4	ILM3	ILM2	ILM1	ILM0	01111_B

The value stored in interrupt level mask register (ILM), is used in the level mask of an interrupt. When the interrupt enable flag (I) is "1", the value of interrupt level mask register (ILM) is compared to the level of the currently requested interrupt. If the value of interrupt level mask register (ILM) is greater (interrupt level is stronger), interrupt requested is accepted. Figure 3.3-6 shows the functions of interrupt level mask.

Figure 3.3-6 Functions of Interrupt Level Mask



The values of interrupt level range from $0(00000_B)$ to $31(11111_B)$. The smaller the value of interrupt level, the stronger it is, and the larger the value, the weaker it is. $0(00000_B)$ is the strongest interrupt level, while $31(11111_B)$ is the weakest.

There are following restrictions on values of the interrupt level mask register (ILM) that can be set from a program.

- When the value of interrupt level mask register (ILM) lies between $0(00000_B)$ to $15(01111_B)$, only values from $0(00000_B)$ to $31(11111_B)$ can be set.
- When the value of interrupt level mask register (ILM) lies between $16(10000_B)$ to $31(11111_B)$, only values between $16(10000_B)$ to $31(11111_B)$ can be set.
- When setting of values between $0(00000_B)$ to $15(01111_B)$ is attempted, 16 is added on automatically and values between $16(10000_B)$ to $31(11111_B)$ are set.

The interrupt level mask register (ILM) is initialized to $15(01111_B)$ following a reset. If an interrupt request is accepted, the interrupt level corresponding to that interrupt is set in the interrupt level mask register (ILM).

For setting a value in interrupt level mask register (ILM) from a program, the STILM instruction is used.

FR81 Family

3.3.6 Condition Code Register (CCR)

Condition code register (CCR) is an 8-bit register that indicates the status of instruction execution. It lies between bit7 to bit0 of the program status (PS).

Figure 3.3-7 shows the bit configuration of condition code register (CCR).

Figure 3.3-7 Condition Code Register (CCR) Bit Configuration

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Initial value
CCR	Reserved	Reserved	S	I	N	Z	V	C	--00XXXX B

The contents of each bit are described below.

[bit7, bit6] Reserved

These are reserved bits. Read value is always "0". Write value should always be "0".

[bit5] S: Stack Flag

This flag selects the stack pointer to be used as general-purpose register R15. When the value of stack flag (S) is "0", system stack pointer (SSP) is used, while when the value is "1", user stack pointer (USP) is used.

Table 3.3-3 Stack Flag (S) of Condition Code Register

flag	value	Meaning
S	0	System stack pointer (SSP)
	1	User stack pointer (USP)

If an EIT operation is accepted, stack flag (S) automatically becomes "0". However, the value of the condition code register (CCR) saved in system stack is the value which is later replaced by "0".

The initial value of stack flag (S) after a reset is "0".

[bit4] I: Interrupt Enable Flag

This flag is used to enable/disable mask-able interrupts. The value "0" of interrupt enable flag (I) disables an interrupt while "1" enables an interrupt. When an interrupt is enabled, the mask operation of interrupt request is performed by interrupt level mask register (ILM).

Table 3.3-4 Interrupt Enable Flag (I) of Condition Code Register

flag	Value	Meaning
I	0	Interrupt disable
	1	Interrupt enable

The value of this flag is replaced by "0" by execution of INT instruction. However, the value of condition code register (CCR) saved in the system stack is the value which is later replaced by "0".

The initial value of an interrupt enable flag (I) after a reset is "0".

[bit3] N: Negative Flag

This flag is used to indicate positive or negative values when the results of a calculation are expressed in two’s complement form. The value "0" of the negative flag (N) indicates a positive value while "1" indicates a negative value.

Table 3.3-5 Negative Flag (N) of Condition Code Register

flag	value	Meaning
N	0	Calculation result is a positive value
	1	Calculation result is a negative value

The initial value of Negative flag (N) after a reset is undefined.

[bit2] Z: Zero Flag

This flag indicates whether the result of a calculation is zero or not. The value "0" of zero flag (Z) indicates a non-zero value, while "1" indicates a zero value.

Table 3.3-6 Zero Flag (Z) of Condition Code Register

flag	value	Meaning
Z	0	Calculation result is a non-zero value
	1	Calculation result is a zero value

The initial value of Zero flag (Z) after a reset is undefined.

[bit1] V: Overflow Flag

This flag indicates whether an overflow has occurred or not when the results of a calculation are expressed in two’s complement form. The value "0" of an overflow flag (V) indicates no overflow, while value "1" indicates an overflow.

Table 3.3-7 Overflow Flag (V) of Condition Code Register

flag	value	Meaning
V	0	No overflow
	1	Overflow

Initial value of overflow flag (V) after a reset is indefinite

[bit0] C: Carry Flag

This flag indicates whether a carry or borrow condition has occurred in the highest bit of the results of a calculation. The value "0" of the carry flag (C) indicates no carry or borrow, while a value "1" indicates a carry or borrow condition.

Table 3.3-8 Carry Flag (C) of Condition Code Register

flag	value	Meaning
C	0	No carry or borrow
	1	Carry or borrow condition

The initial value of a carry flag (C) after reset is undefined.

FR81 Family

3.3.7 System Condition Code Register (SCR)

System condition code register (SCR) is a 3-bit register used to control the intermediate data of stepwise division and step trace trap. It lies between bit10 to bit8 of the program status (PS).

Figure 3.3-8 shows the bit configuration of system condition code register (SCR).

Figure 3.3-8 System Condition Code Register (SCR) Bit Configuration

bit10	bit9	bit8	Initial value
D1	D0	T	XX0B

The contents of each bit are described below.

[bit10, bit9] D1, D0: Step Intermediate Data

These bits are used for intermediate data in stepwise division. This register is used to assure resumption of division calculations when the stepwise division program is interrupted during processing.

If changes are made to the contents of the intermediate data (D1, D0) during division processing, the results of the division are not assured. If another processing is performed during stepwise division processing, division can be resumed by saving/retrieving the program status (PS) in/from the system stack.

Intermediate data (D1, D0) of stepwise division is made into a set by referencing the dividend and divisor by executing the "DIV0S" instruction. It is cleared by executing the "DIV0U" instruction.

The initial value of intermediate data (D1, D0) of stepwise division after a CPU reset is undefined.

[bit8] T: Step Trace Trap Flag

This flag specifies whether the step trace trap operation has to be enabled or not. When the step trace trap flag (T) is set to "1", step trace flag operation is enabled and the CPU generates an EIT event by trap operation after each instruction execution.

Table 3.3-9 Step Trace Trap Flag (T) of System Condition Code Register

flag	value	Meaning
T	0	Step trace trap disabled
	1	Step trace trap enabled

When the step trace trap flag (T) is "1", all NMI & user interrupts are disabled.

Step trace trap function uses an emulator. During a user program which uses the emulator, step trace trap function cannot be used (the emulator cannot be used for debugging in the step trace trap routine).

The initial value of step trace trap flag (T) after a reset is "0".

3.3.8 Return Pointer (RP)

Return pointer (RP) is a 32-bit register which stores the address for returning from a subroutine. It stores the program counter (PC) value upon execution of a CALL instruction.

Figure 3.3-9 shows the bit configuration of return pointer (RP).

Figure 3.3-9 Return Pointer (RP) Bit Configuration

In case of a CALL instruction with a delay slot, the value stored in RP will be the address of the CALL instruction +4.

In case of a CALL instruction without a delay slot, the value stored in RP will be the address of the CALL instruction +4.

When returning from a subroutine by the RET instruction, the address stored in the return pointer (RP) is returned to the program counter (PC).

Return pointer (RP) does not have a stack configuration. When calling another subroutine from the subroutine called using the CALL instruction, it is necessary to first save the contents of the return pointer (RP) and restore them before executing the RET instruction.

Figure 3.3-10 shows a sample operation of the return pointer (RP) during the execution of a CALL instruction without a delay slot, and Figure 3.3-11 shows a sample operation of return pointer (RP) during the execution of a RET instruction.

Figure 3.3-10 Sample Operation of RP during Execution of a CALL Instruction without a Delay Slot

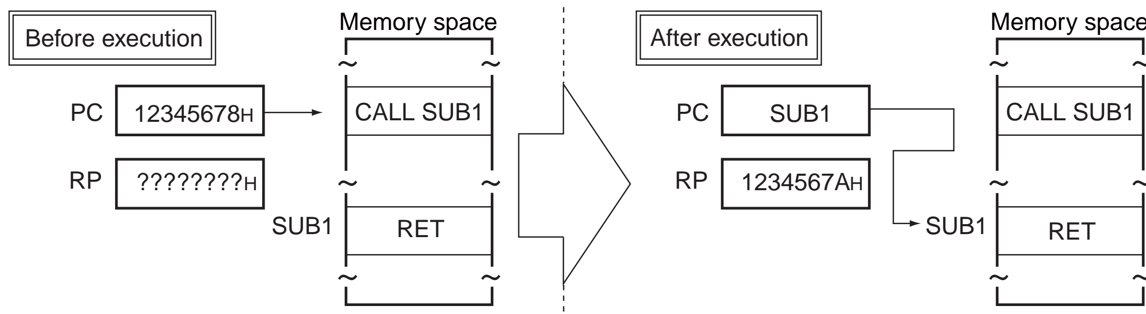
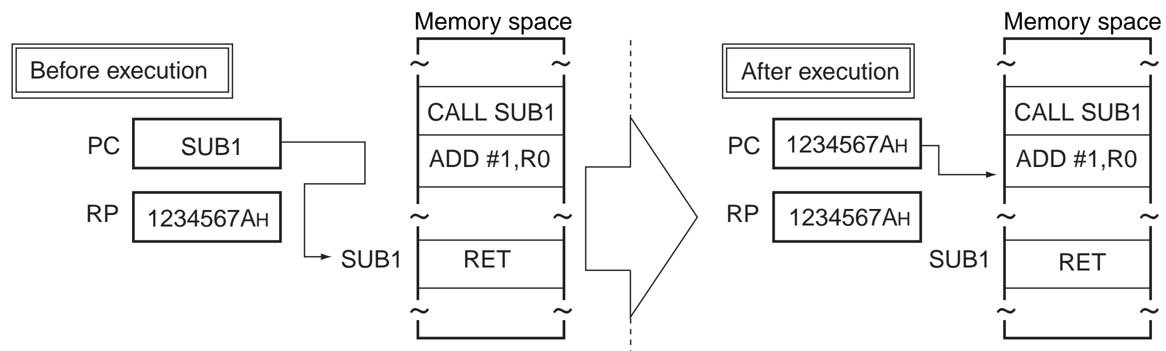


Figure 3.3-11 Sample Operation of RP during Execution of a RET Instruction.



3.3.9 System Stack Pointer (SSP)

The system stack pointer (SSP) is a 32-bit register that indicates the address to be saved/restored to the system stack used at the time of EIT processing. The system stack pointer (SSP) is available when CPU is in privilege mode (UM=0).

Figure 3.3-12 shows the bit configuration of system stack pointer (SSP).

Figure 3.3-12 System Stack Pointer (SSP) Bit Configuration

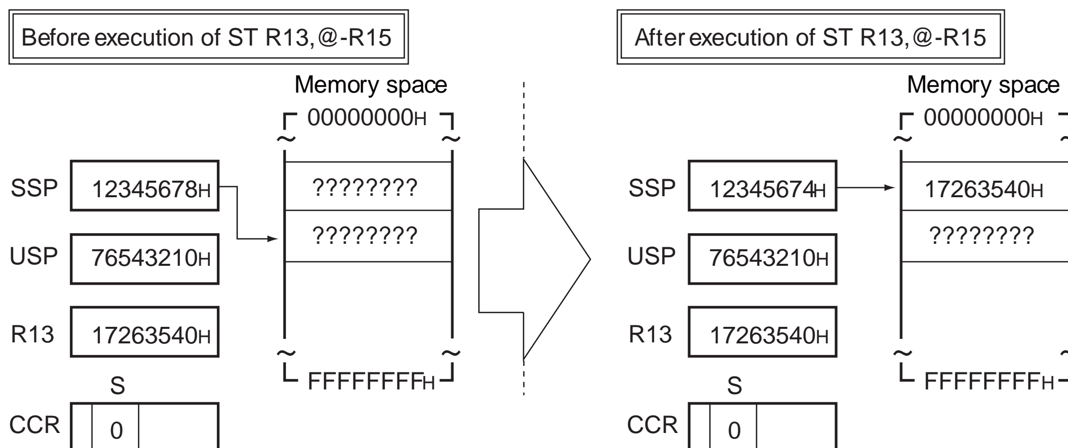
When the stack flag (S) in the condition code register (CCR) is "0", the general-purpose register R15 is used as the system stack pointer (SSP). In a normal instruction, system stack pointer is used as the general-purpose register R15.

When an EIT event occurs, regardless of the value of the stack flag (S), the program counter (PC) and program status (PS) values are saved to the system stack area designated by system stack pointer (SSP). The value of stack flag (S) is stored in the system stack as program status (PS), and is restored from the system stack at the time of returning from the EIT event using RETI instruction.

System stack uses pre-decrement/post-decrement for storing and retrieving data. While saving data, after performing a data size decrement on the value of system stack pointer (SSP), it is written onto the address indicated by system stack pointer (SSP). While retrieving data, after the data is read from the address indicated by the system stack pointer (SSP), a data size increment is performed on the value of system stack pointer (SSP).

Figure 3.3-13 shows an example of system stack pointer (SSP) operation while executing instruction "ST R13,@-R15" when the stack flag (S) is set to "0".

Figure 3.3-13 Example of System Stack Pointer (SSP) Operation



3.3.10 User Stack Pointer (USP)

User stack pointer (USP) is a 32-bit register used to save/retrieve data to/from the user stack. The user stack pointer (USP) is available irrespective of the CPU: whether it is in privilege mode (UM=0) or in user mode (UM=1). In privilege mode, the stack pointer should be selected by rewriting the stack flag (S). In user mode, only the user stack pointer (USP) is available.

Figure 3.3-14 shows the bit configuration of user stack pointer (USP).

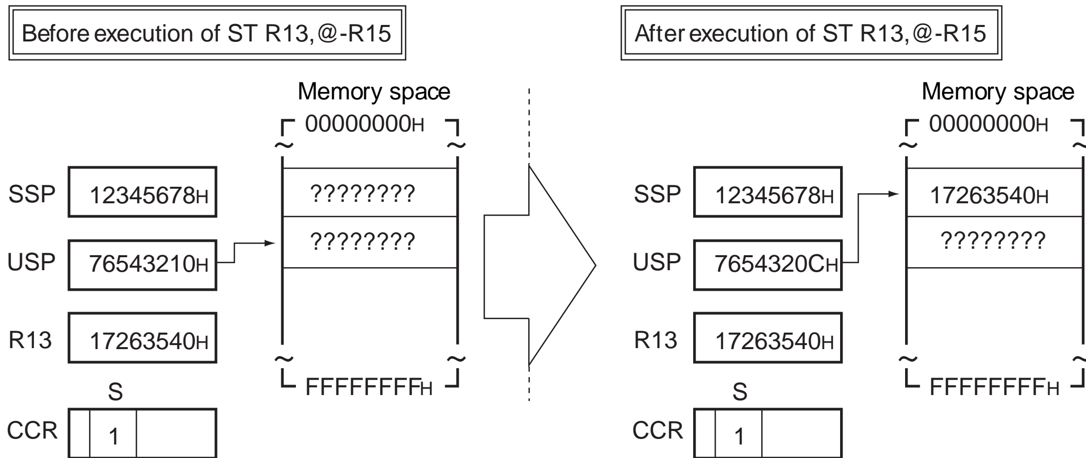
Figure 3.3-14 User Stack Pointer (USP) Bit Configuration

When the stack flag (S) in the condition code register (CCR) is "1", the general-purpose register R15 is used as the user stack pointer (USP). In a normal instruction, user stack pointer (USP) is used as the general-purpose register R15.

User stack uses pre-decrement/post-decrement to save/retrieve data. While saving data, after performing a data size decrement on the value of user stack pointer (USP), it is written onto the address indicated by the user stack pointer (USP). While retrieving data, the data is read from the address indicated by the user stack pointer (USP), and a data size increment is performed on the value of user stack pointer (USP).

Figure 3.3-15 shows an example of user stack pointer (USP) operation while executing the instruction "ST R13,@-R15" when the stack flag (S) is set to "1".

Figure 3.3-15 Example of User Stack Pointer (USP) Operation



3.3.11 Table Base Register (TBR)

Table base register (TBR) is a 32-bit register that designates the vector table containing the entry addresses for EIT operations.

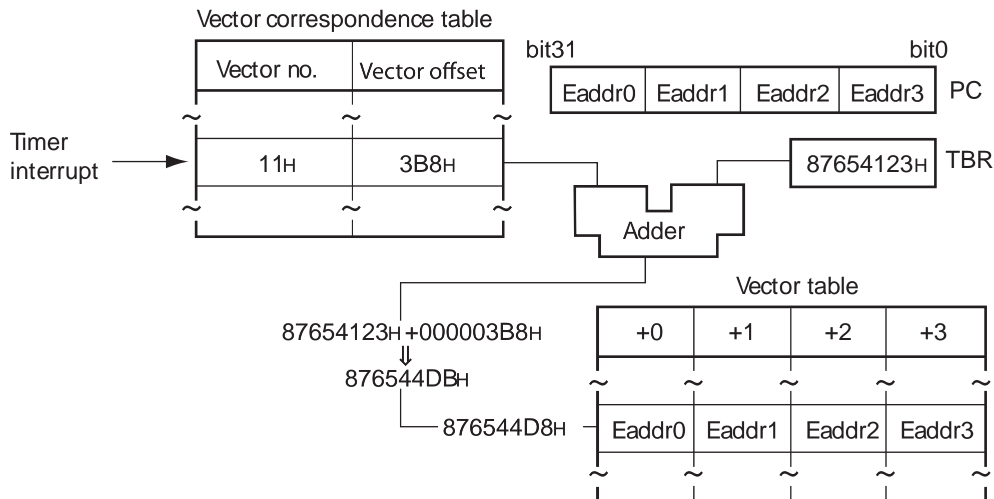
Figure 3.3-16 shows the bit configuration of table base register (TBR).

Figure 3.3-16 Table Base Register (TBR) Bit Configuration

The address of the reference vector is determined by the sum of the contents of the table base register (TBR) and the vector offset corresponding to the EIT operation generated. Vector table layout is realized in word units. As the address of the calculated vector is in word units, the lower two bits of the resulting address value are explicitly read as “0”.

Figure 3.3-17 shows an example of table base register (TBR).

Figure 3.3-17 Example of Table Base Register (TBR) Example



The reset value of table base register (TBR) is 000F $FC00_H$. Do not set a value above FFFF $FC00_H$ for the table base register (TBR).

Precautions:

If values greater than FFFF $FC00_H$ are assigned to the table base register (TBR), this operation may result in an overflow when summed with the offset value. An overflow in turn will result in vector access to the area 0000 0000_H to 0000 $03FF_H$, which can cause a program run away.

3.3.12 Multiplication/Division Register (MDH, MDL)

Multiplication/Division register (MDH, MDL) is a 64-bit register comprised of MDH represented by the higher 32 bits and MDL represented by the lower 32 bits. During multiplication, the product is stored. During division, the value set for the dividend and the quotient is stored.

Figure 3.3-18 shows the bit configuration of Multiplication/Division register (MDH, MDL).

Figure 3.3-18 Multiplication/Division Register (MDH, MDL) Bit Configuration

The function of Multiplication/Division register (MDH, MDL) is different during a multiplication and during a division operation.

● Function during Multiplication

In case of a 32 bit × 32 bit multiplication (MUL, MULU instruction), the calculation result of 64-bit length is stored in the product register (MDH, MDL) as follows.

MDH: higher 32 bits

MDL: lower 32 bits

In case of a 16 bit × 16 bit multiplication (MULH, MULUH instruction), the calculation result of 32-bit length is stored in the product register (MDH, MDL) as follows.

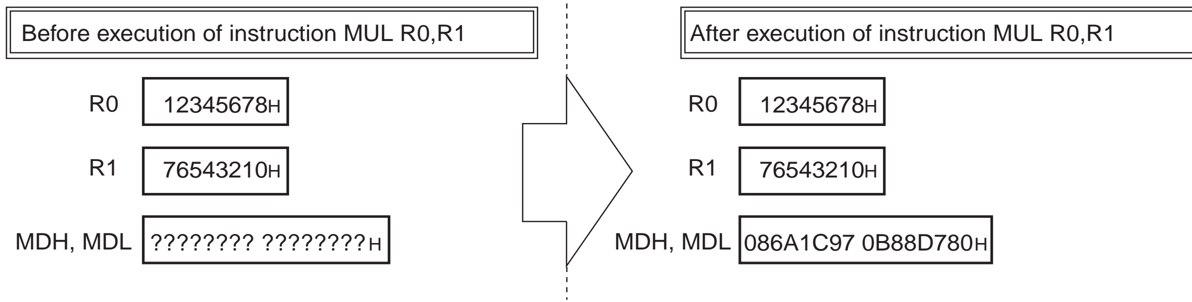
MDH: undefined

MDL: result 32 bits

Figure 3.3-19 shows an example of multiplication operation using Multiplication/Division register (MDH, MDL).

FR81 Family

Figure 3.3-19 Example of Multiplication Operation using Multiplication/Division Register (MDH, MDL)



● Function during Division

Before starting the calculation, the dividend is stored in the Multiplication/Division register (MDH, MDL).

MDH: don’t care

MDL: dividend

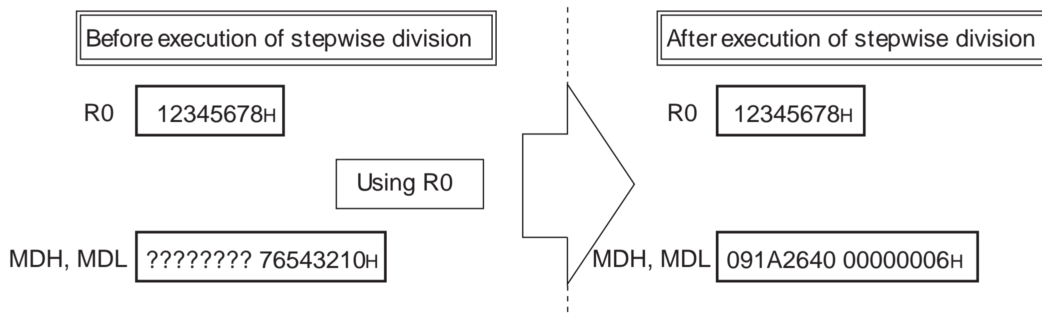
When division is performed using any of the instructions DIV0S/DIV0U, DIVI1, DIV2, DIV3, DIV4S meant for division, the result of division is stored in the Multiplication/Division register (MDH, MDL) as follows.

MDH: remainder

MDL: quotient

Figure 3.3-20 shows an example of division operation using Multiplication/Division register (MDH, MDL).

Figure 3.3-20 Example of Division Operation using Multiplication/Division Register (MDH,MDL)



3.3.13 Base Pointer (BP)

The base pointer (BP) register is used for pointing in base pointer indirect addressing mode.

Figure 3.3-21 shows the bit configuration of base pointer (BP).

Figure 3.3-21 Base Pointer (BP) Bit Configuration

3.3.14 FPU Control Register (FCR)

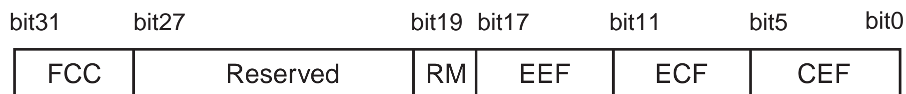
FPU control register (FCR) is a 32-bit register used to control the FPU. It has a flag that indicates the settings and status of the FPU operation mode.

The FPU control register (FCR) consists of the following five parts:

- Floating point condition code (FCC)
- Rounding mode (RM)
- Floating point exception enable flag (EEF)
- Floating point exception accumulative flag (ECF)
- Floating point exception flag (CEF)

Figure 3.3-22 shows the bit configuration of FPU control register (FCR).

Figure 3.3-22 FPU control register (FCR) Bit Configuration



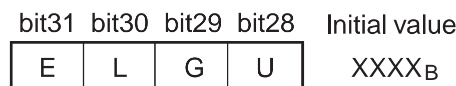
The reserved bits of the FPU control register (FCR) are all reserved for future expansion. The read value of reserved bits is always "0". The write value should always be "0".

■ Floating point condition code (FCC)

Floating point condition code (FCC) is a 4-bit register that stores the condition code of a floating point calculation result. It lies between bit 31 and bit 28 of the FPU control register (FCR).

Figure 3.3-23 shows the bit configuration of the floating point condition code (FCC).

Figure 3.3-23 Floating point condition code (FCC) Bit Configuration



FR81 Family

The content of each bit are described below.

[bit31] E : E flag

This flag indicates that FRj and FRi are equal based on the floating point compare instruction (FCMP) results.

[bit30] L : L flag

This flag indicates that FRi is less than FRj based on the floating point compare instruction (FCMP) results.

[bit29] G : G flag

This flag indicates that FRi is greater than FRj based on the floating point compare instruction (FCMP) results.

[bit28] U : U flag

This flag indicates that no comparison can be made (Unordered) based on the floating point compare instruction (FCMP) results.

■ Rounding mode (RM)

Rounding mode (RM) is a 2-bit register that designates rounding mode of floating point calculation results. It lies between bit 19 and bit 18 of the FPU control register (FCR). In FR81 Family CPU, only rounding up to the nearest value (RM=00_B) can be set.

Figure 3.3-24 shows the bit configuration of rounding mode (RM). Table 3.3-10 shows details of the rounding mode.

Figure 3.3-24 Rounding mode (RM) Bit Configuration

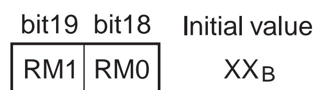


Table 3.3-10 Rounding mode

RM	Rounding mode
00_B	The nearest value
01_B	0
10_B	+∞
11_B	-∞

■ Floating point exception enable flag (EEF)

Floating point exception enable flag (EEF) is a 6-bit register that enables exception occurrences of floating point calculation. It lies between bit 17 and bit 12 of the FPU control register (FCR).

Figure 3.3-25 shows the bit configuration of the floating point exception enable flag (EEF).

Figure 3.3-25 Floating point exception enable flag (EEF) Bit Configuration

bit17	bit16	bit15	bit14	bit13	bit12	Initial value
D	X	U	O	Z	V	$XXXXXX_B$

The content of each bit are described below.

[bit17] D : D flag

This is a unnormalized number input exception enable flag. When this bit has been set to "1", the FPU exception occurs upon input of an unnormalized number. When this bit has been set to "0", the unnormalized number is regarded as "0" for calculation purposes.

[bit16] X : X flag

This is an inexact exception enable flag. When this bit is set to "1" and an inexact has occurred in the calculation result, FPU exception occurs. When this bit is set to "0", the value resulting from rounding up is written in the register.

[bit15] U : U flag

This is an underflow exception enable flag. When this bit is set to "1" and an underflow has occurred in the calculation result, FPU exception occurs. When this bit is set to "0", a value "0" is written in the register.

[bit14] O : O flag

This is an overflow exception enable flag. When this bit is set to "1" and an overflow has occurred in the calculation result, FPU exception occurs. When this bit is set to "0", $\pm \infty$ or $\pm$ MAX is written in the register in accordance with the rounding mode (RM).

[bit13] Z : Z flag

This is a division-by-zero exception enable flag. When this bit is set to "1" and division-by-zero is carried out, FPU exception occurs. When this bit is set to "0", infinite (∞), which indicates that the calculation has been carried out appropriately, is written in the register.

[bit12] V : V flag

This is an invalid calculation exception enable flag. When this bit is set to "1" and an invalid calculation is carried out, FPU exception occurs When this bit is set to "0", QNaN is written in the register in the calculation type instruction, $\pm$ MAX is written in the register in the conversion instruction, and "1" (unordered) is set for the U flag of the floating point condition code (FCC) in the compare instruction.

■ Floating point exception accumulative flag (ECF)

Floating point exception accumulative flag (ECF) is a 6-bit register that indicates the accumulative number of occurrences of floating point calculation exceptions. It lies between bit 11 and bit 6 of the FPU control register (FCR). Only a "0" can be written in the accumulative flags. The flag value will not be changed when "1" is written in the accumulative flags. The write value is evaluated by bit.

Figure 3.3-26 shows the bit configuration of the floating point exception accumulative flag (ECF).

Figure 3.3-26 Floating point exception accumulative flag (ECF) Bit Configuration

bit11	bit10	bit9	bit8	bit7	bit6	Initial value
D	X	U	O	Z	V	$XXXXXX_B$

The content of each bit are described below.

[bit11] D : D flag

This flag indicates that an unnormalized number has been entered while the unnormalized number input exception is disabled (EEF:D=0). This is a accumulative flag.

[bit10] X : X flag

This flag indicates that the calculation result has become inexact while the inexact exception is disabled (EEF:X=0). This is a accumulative flag.

[bit9] U : U flag

This flag indicates that an underflow has occurred in the calculation result while the underflow exception is disabled (EEF:U=0). This is a accumulative flag.

[bit8] O : O flag

This flag indicates that an overflow has occurred in the calculation result while the overflow exception is disabled (EEF:O=0). This is a accumulative flag.

[bit7] Z : Z flag

This flag indicates that a division by zero has occurred while the division-by-zero exception is disabled (EEF:Z=0). This is a accumulative flag.

[bit6] V : V flag

This flag indicates that an invalid calculation has been carried out while the invalid calculation exception is disabled (EEF:V=0). This is a accumulative flag.

■ Floating point exception flag (CFE)

Floating point exception flag (CFE) is a 6-bit register that indicates the exception occurrence of floating point calculation. It lies between bit 5 and bit 0 of the FPU control register (FCR). Each flag is set according to the calculation result. Each flag shall be cleared using software. Each flag can be set only to "0", and writing "1" to the flag is invalid. The write value is evaluated by bit. If the flag has not been cleared during exception processing, each flag is cumulated.

Figure 3.3-27 shows the bit configuration of the floating point exception flag (CFE).

Figure 3.3-27 Floating point exception flag (CFE) Bit Configuration

bit5	bit4	bit3	bit2	bit1	bit0	Initial value
D	X	U	O	Z	V	$XXXXXX_B$

The content of each bit are described below.

[bit5] D : D flag

This flag is set when an unnormalized number has been input while the unnormalized number input exception is enabled (EEF:D=1).

[bit4] X : X flag

This flag is set when the calculation result has become inexact while the inexact exception is enabled (EEF:X=1).

[bit3] U : U flag

This flag is set when an underflow has occurred in the calculation result while the underflow exception is enabled (EEF:U=1).

[bit2] O : O flag

This flag is set when an overflow has occurred in the calculation result while the overflow exception is enabled (EEF:O=1).

[bit1] Z : Z flag

This flag is set when a division by zero has occurred while the division-by-zero exception is enabled (EEF:Z=1).

[bit0] V : V flag

This flag is set when an invalid calculation has been carried out while the invalid calculation exception is enabled (EEF:V=1).

3.3.15 Exception status register (ESR)

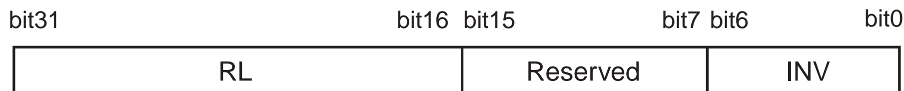
This is a 32-bit register that indicates the balance of process when an exception occurs while executing the invalid instruction exception source and the multiple load/store instruction.

The exception status register (ESR) consists of the following two parts:

- Register list (RL)
- Invalid instruction exception source (INV)

Figure 3.3-28 shows the bit configuration of the exception status register (ESR).

Figure 3.3-28 Exception status register (ESR) Bit Configuration



The reserved bits of the exception status register (ESR) are all reserved for future expansion. The read value of reserved bits is always "0". Write value should always be "0".

■ Register List (RL)

Register list (RL) is a 16-bit register that indicates registers whose transmission has not ended when an exception occurs while a LDM0, LDM1, STM0, STM1, FLDM, or FSTM instruction is executed. It lies between bit 31 and bit 16 of the exception status register (ESR). The register list (RL) value is updated only when an exception occurs while a LDM0, LDM1, STM0, STM1, FLDM, or FSTM instruction is executed.

Figure 3.3-29 shows the bit configuration of the register list (RL), and Table 3.3-11 shows the correspondence between the register list (RL) bits and the registers.

Figure 3.3-29 Register List (RL) Bit Configuration

Table 3.3-11 Correspondence between the register list (RL) bits and the registers

bit of ESR register	31	30	29	28	27	26	25	24
RL bit	RL15	RL14	RL13	RL12	RL11	RL10	RL9	RL8
LDM1, LDM0 instruction	R15	R14	R13	R12	R11	R10	R9	R8
STM1, STM0 instruction	R0	R1	R2	R3	R4	R5	R6	R7
FLDM instruction	FR15	FR14	FR13	FR12	FR11	FR10	FR9	FR8
FSTM instruction	FR0	FR1	FR2	FR3	FR4	FR5	FR6	FR7

ESR bit	23	22	21	20	19	18	17	16
RL bit	RL7	RL6	RL5	RL4	RL3	RL2	RL1	RL0
LDM1, LDM0 instruction	R7	R6	R5	R4	R3	R2	R1	R0
STM1, STM0 instruction	R8	R9	R10	R11	R12	R13	R14	R15
FLDM instruction	FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
FSTM instruction	FR8	FR9	FR10	FR11	FR12	FR13	FR14	FR15

■ Invalid instruction exception source (INV)

Invalid instruction exception source (INV) is a 7-bit register that indicates the source causing an invalid instruction exception. It lies between bit 6 and bit 0 of the exception status register (ESR). Each flag is set only when the source occurs. Each flag shall be cleared using software. Each flag can be set only to "0", and writing "1" to the flag is invalid. The write value is evaluated by bit.

Figure 3.3-30 shows the bit configuration of the invalid instruction exception source (INV).

Figure 3.3-30 Invalid instruction exception source (INV) Bit Configuration

bit6						bit0	Initial value
DT	IF	FPU	PI	SPR	DS	RI	0000000_B

The content of each bit are described below.

[bit6] DT : Data access error

This flag is set when a bus error occurs during data access to a buffer-disabled area, or a system register is accessed in user mode.

[bit5] IF : Instruction fetch error

This flag is set when a bus error occurs during instruction fetch, and the instruction is executed.

[bit4] FPU : FPU absence error

This flag is set when an floating point type instruction is executed on a model without FPU installed.

[bit3] PI : Privilege instruction execution

This flag is set when a RETI or STILM instruction is executed in user mode.

[bit2] SPR : System-dedicated register access

This flag is set when a MOV or LD instruction is executed to the table base register (TBR), system stack pointer (SSP), or the exception status register (ESR) in user mode.

[bit1] DS : Invalid instruction placement on delay slot

This flag is set when an instruction that cannot be placed on delay slot is executed on the delay slot.

[bit0] RI : Undefined instruction

This flag is set when an undefined instruction code is being executed.

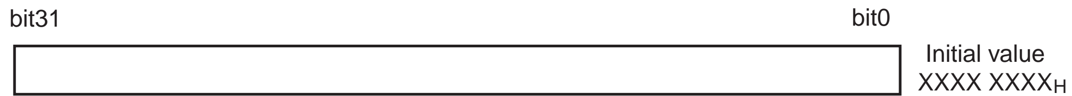
FR81 Family

3.3.16 Debug Register (DBR)

The debug register (DBR) is a dedicated register accessible only in the debug state. Writing to this register other than in debug state is regarded as invalid.

Figure 3.3-31 shows the bit configuration of Debug Register (DBR).

Figure 3.3-31 Debug Register (DBR) Bit Configuration



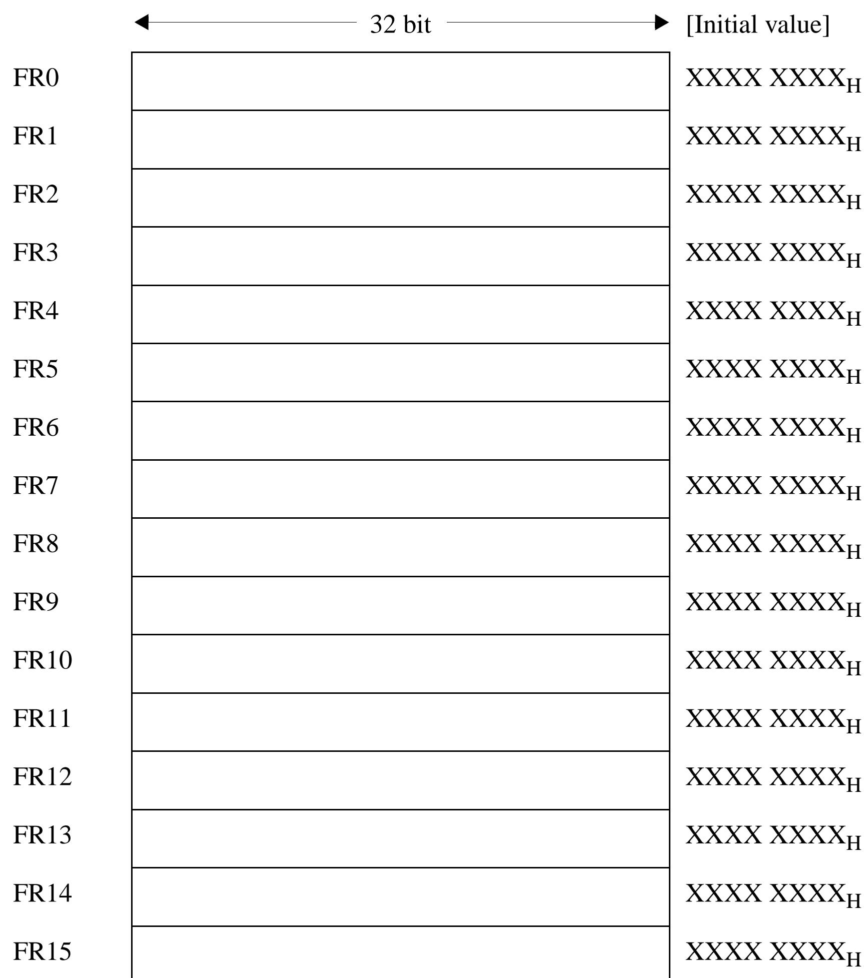
3.4 Floating-point Register

Floating point registers are using that store results for floating point calculations.

The floating-point register is 16, each having 32-bit length. As for the register, the name of FR0 to FR15 is named.

Figure 3.4-1 shows the construction and the initial value of the floating-point register.

Figure 3.4-1 The construction and the initial value of the floating-point register



CHAPTER 4 RESET AND "EIT" PROCESSING

This chapter describes reset and EIT processing in the FR81 family CPU. EIT processing is the generic name for exceptions, interrupt and trap.

4.1 Reset

4.2 Basic Operations in EIT Processing

4.3 Processor Operation Status

4.4 Exception Processing

4.5 Interrupts

4.6 Traps

4.7 Multiple EIT processing and Priority Levels

4.8 Timing When Register Settings Are Reflected

4.9 Usage Sequence of General Interrupts

4.10 Precautions

4.1 Reset

A reset forcibly terminates the current process, initializes the device, and restarts the program from the reset vector entry address.
The reset process is executed in privilege mode. Transition to user mode should be carried out by executing a RETI instruction.

When a reset is generated, CPU terminates the processing of the instruction execution at that time and goes into inactive status until the reset is cancelled. When the reset is cancelled, the CPU initializes all internal registers and starts execution beginning with the program indicated by the new value of the program counter (PC).

Reset processing has a higher priority level than each operation of the EIT processing described later. Reset is accepted even in between an EIT processing.

When a reset is generated, FR81 family CPU makes an attempt to initialize each register, but all registers cannot be initialized. Each register sets a value through the program executed after a reset, and uses it. Table 4.1-1 shows the registers that are initialized following a reset.

Table 4.1-1 Registers that are initialized following a reset

Register	Initial Value	Remarks
Program counter (PC)	Word data at location $000F\ FFFC_H$	Reset vector
Interrupt level mask register (ILM)	$15(01111_B)$	
Step trace trap flag (T)	“0”	Trace OFF
Interrupt enable flag (I)	“0”	Interrupt disabled
Stack flag (S)	“0”	Use SSP
Table base register (TBR)	$000F\ FC00_H$	
System stack pointer (SSP)	$0000\ 0000_H$	
Debug state flag (DBG)	“0”	No debug state
User mode flag (UM)	“0”	Privilege mode
Exception status register (ESR)	$0000\ 0000_H$	
General-purpose register R15	SSP	As per stack flag (S)

For details of My computer built-in functions (peripheral devices, etc.) following a reset, refer to the Hardware Manual provided with each device.

4.2 Basic Operations in EIT Processing

Exceptions, interrupts and traps are similar operations applied under partially different conditions. They save information for terminating or restarting the execution of instructions and perform branching to a processing program.

4.2.1 Types of EIT Processing and Prior Preparation

EIT processing is a method which terminates the currently executing process and transfers control to a predetermined processing program after saving restart information to the memory. EIT processing programs can return to the prior program by use of the RETI instruction.

EIT processing operates in essentially the same manner for exceptions, interrupts and traps, with a few minor differences listed below by which it differentiates them.

- Exceptions are related to the instruction sequence, and processing is designed to resume from the instruction in which the exception occurred.
- Interrupts originate independently of the instruction sequence. Processing is designed to resume from the instruction immediately following the acceptance of the interrupt.
- Traps are also related to the instruction sequence, and processing is designed to resume from the instruction immediately following the instruction in which the trap occurred.

While performing EIT processing, apply to the following prior settings in the program.

- Set the values in vector table (defining as data)
- Set the value of system stack pointer (SSP)
- Set the value of table base register (TBR) as the initial address in the vector table
- Set the value of interrupt level mask register (ILM) above 16(10000_B)
- Set the interrupt enable flag (I) to "1"

The setting of interrupt level mask register (ILM) and interrupt enable flag (I), will be required at the time of using interrupts.

To support the emulator debugger debug function, a processing called "break" is carried out in the user state of debugging. The processing differs from usual EIT. The following shows the sources causing "break". The break processing is executed when the source is detected in the user state.

- Instruction break exception
- Break interrupt
- Step trace trap
- INTE instruction execution

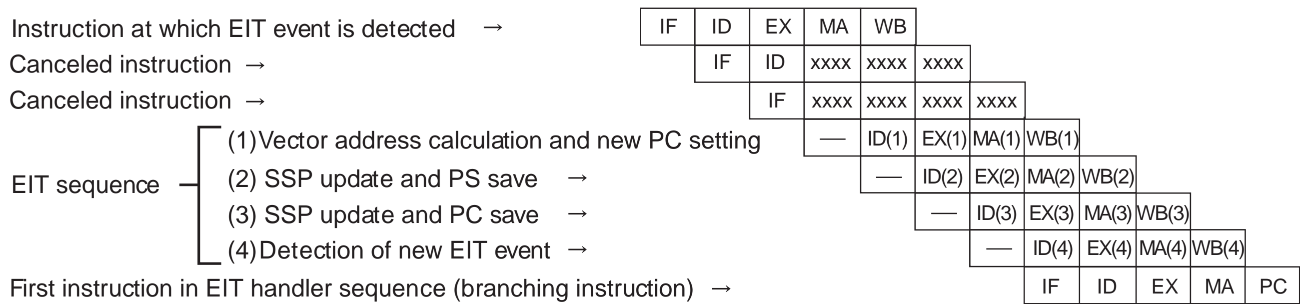
4.2.2 EIT Processing Sequence

FR81 family CPU processes EIT events as follows.

1. The vector table indicated by the table base register (TBR) and the offset value of the vector number corresponding to the particular EIT are used to determine the entry address for the processing program for the EIT.
2. For restarting, the contents of the old program counter (PC) and the old program status (PS) are saved to the stack area designated by the system stack pointer (SSP).
3. "0" is saved in the stack flag (S). Also, the interrupt level Mask Register (ILM) and interrupt enable flag (I) are updated through EIT.
4. Entry address is saved in the program counter (PC).
5. After the processing flow is completed, just before the execution of the instruction in the entry address, the presence of new EIT sources is determined.

Figure 4.2-1 shows the operations in the EIT processing sequence.

Figure 4.2-1 Operations in EIT Processing Sequence



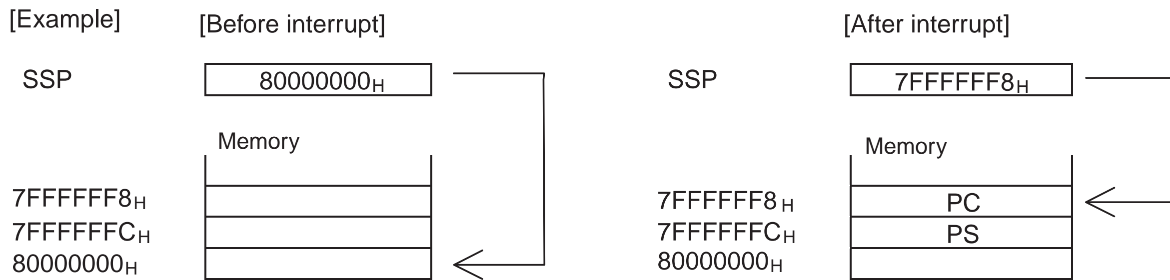
Vector tables are located in the main memory, occupying an area of 1 Kbyte beginning with the address shown in the table base register (TBR). This area is used as a table of entry addresses for EIT processing.

For details on vector tables, refer to "2.1.2 Vector Table Area" and "3.3.6 Condition Code Register (CCR)".

Regardless of the value of stack flag (S), the program status (PS) and program counter (PC) is saved in the stack pointed to by the system stack pointer (SSP). After an EIT processing has commenced, the program counter (PC) is saved in the address pointed to by the system stack pointer (SSP), while the program status (PS) is saved at address 4 plus the address pointed to by the system stack pointer (SSP).

Figure 4.2-2 shows an example of saving program counter (PC) and program status (PS) during the occurrence of an EIT event.

Figure 4.2-2 Example of storing of PC, PS during an EIT event occurrence



4.2.3 Recovery from EIT Processing

RETI instruction is used for recovery from an EIT processing program. The RETI instruction retrieves the value of program counter (PC) and program status (PS) from the system stack, EIT and recovers from the EIT processing.

1. Retrieving program counter (PC) from the system stack
   (SSP) → PC   SSP+4 → SSP
2. Retrieving program status (PS) from the system stack
   (SSP) → PS   SSP+4 → SSP

To ensure the program execution results after recovery from the EIT processing program, it is required that all contents of the CPU registers before the commencement of EIT processing program have been saved at the time of recovery. The registers used in the EIT processing programs should be saved in the system stack and retrieved just before the RETI instruction.

4.3 Processor Operation Status

Processor operation is comprised of four states: Reset, normal operation, low-power consumption, and debugging.

● Reset state

A state where the CPU is being reset. Two levels are provided for the reset state: Initialize level and reset level. When an initialize level reset is issued, all functions inside the MCU chip are initialized. When a reset level is issued, functions except debug control, and some parts of the clock and reset controls are initialized.

● Normal operation state

A state where the sequential instructions and EIT processing are currently executed. Privilege mode (UM=0) and user mode (UM=1) are provided for the normal operation state. Some instructions and access destinations are disabled in user mode while they are enabled in privilege mode.

After release of a reset state, the system enters privilege mode in the normal operation state, and is shifted to user mode by executing a RETI instruction. In the normal operation state, user mode is shifted to privilege mode by executing reset or EIT, and privilege mode is shifted to user mode by executing a RETI instruction.

● Low-power consumption stat

A state where the CPU stops operating to save power consumption. Transition to the lower power consumption state is carried out by controlling the stand-by in the clock control section. Three modes are provided for the low-power consumption state: Sleep, stop and clock. An interrupt shall be used to restore the system from the low-power consumption state.

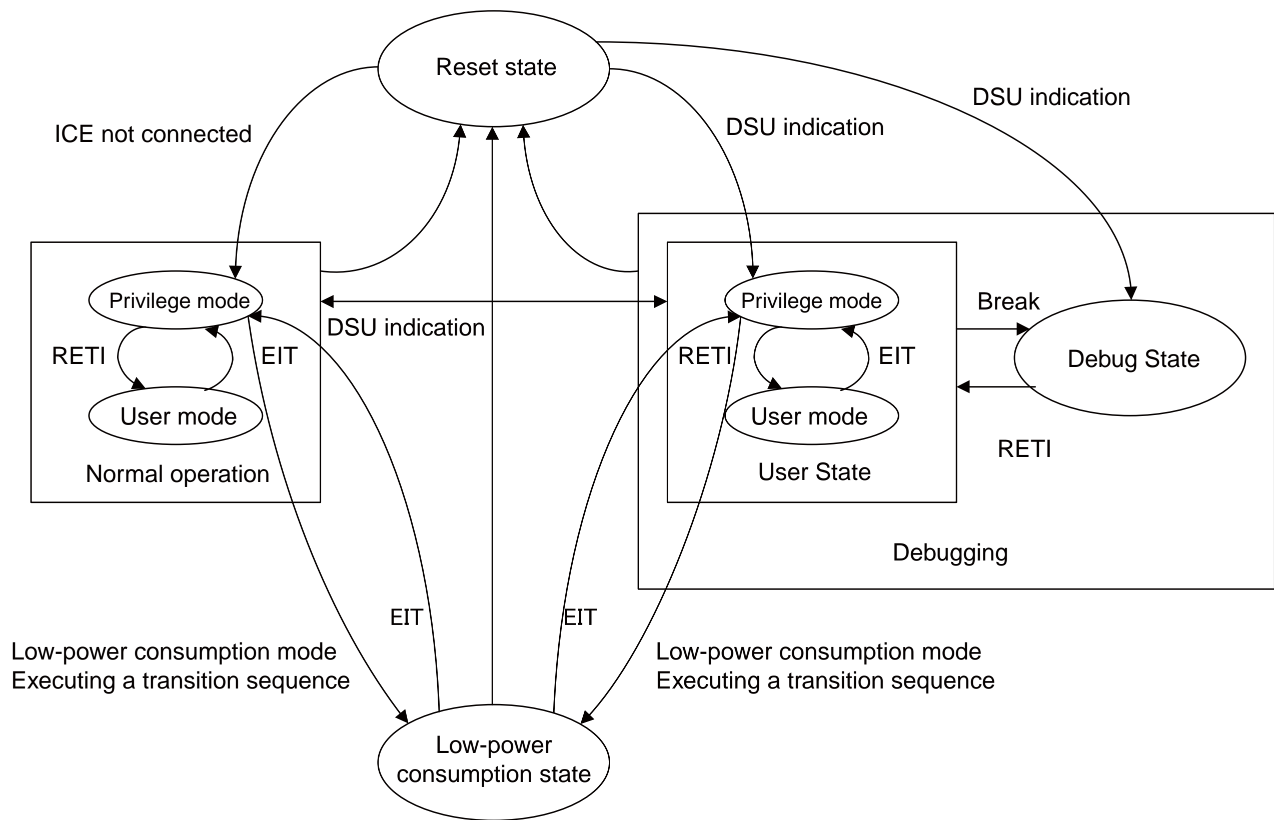
● Debugging state

A state where an in-circuit emulator (ICE) is connected, and debug related functions are enabled. The debugging state is separated into a user state and a debug state. In principle, a debugging state shall be shifted to the other state via a reset. However, a normal operation state can be forcibly shifted to a debugging state.

As is the case with the normal operation state, privilege mode (UM=0) and user mode (UM=1) are provided for the user state. However, when a break is executed for debugging the state is shifted to the debug state. It is carried out in privilege mode under the debug state, and all registers and whole memory area can be accessed by disabling the memory protection and other functions. The debug state is shifted to the user state by executing a RETI instruction.

Figure 4.3-1 shows transition between the processor operation states.

Figure 4.3-1 Transition between processor operation states



4.4 Exception Processing

Exceptions originate when there is a problem in the instruction sequence. Exceptions are processed by first saving the necessary information to resume the currently executing instruction, and then starting the processing routine corresponding to the type of exception that has occurred.

Branching to the exception processing routine takes place before execution of the instruction that has caused the exception. The address of the instruction in which the exception occurs becomes the program counter (PC) value that is saved to the stack at the time of occurrence of the exception.

The following factors can cause occurrence of an exception:

- Invalid instruction exception
- Instruction access protection violation exception
- Data access protection violation exception
- FPU exception
- Instruction break
- Guarded access break

4.4.1 Invalid Instruction Exception

An invalid instruction exception occurs when an invalid instruction is being executed. The following sources can cause the invalid instruction exception.

- Executing an undefined instruction code.
- Executing on delay slot an instruction that cannot be placed on the delay slot.
- Writing to a system-dedicated register (TBR, SSP, or ESR) in user mode (with MOV or LD instruction).
- Executing a privilege instruction (RETI or STILM) in user mode.
- Executing a floating point instruction while FPU is absent.
- Occurrence of a bus error during instruction fetch.
- Occurrence of a bus error or violation of system register access during data access to a buffer-disabled area.

The following operations are performed if an invalid-instruction exception is accepted.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   "0" → UM "0" → S
2. Contents of program status (PS) are saved to the system stack.
   SSP - 4 → SSP PS → (SSP)
3. Contents of the program counter (PC) of an exception source instruction are saved to the system stack.
   SSP - 4 → SSP PC → (SSP)

4. The program counter (PC) value is updated by referring to the vector table.
   $(TBR + 3C4_H) \rightarrow PC$
5. A new EIT event is detected.

The address saved to the system stack as a program counter (PC) value represents the instruction itself that caused the undefined instruction exception. When a RETI instruction is executed, the contents of the system stack should be rewritten with the exception processing routine so that the execution will either resume from the address of the instruction next to the instruction that caused the exception.

4.4.2 Instruction Access Protection Violation Exception

An instruction access protection exception occurs when an instruction is executed in an area protected by the memory protection function.

During debugging, this exception can be treated as a break source according to an indication from the debugger. In this case, the instruction access protection violation exception does not occur.

Upon acceptance of the instruction access protection violation exception, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   "0" → UM "0" → S
2. The contents of the program status (PS) are saved to the system stack.
   SSP - 4 → SSP PS → (SSP)
3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.
   SSP - 4 → SSP PC → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.
   $(TBR + 3E4_H) \rightarrow PC$
5. A new EIT event is detected.

4.4.3 Data Access Protection Violation Exception

A data access protection violation exception occurs when an invalid data access is executed in an area protected by the memory protection function.

During debugging, this exception can be treated as a break source according to an indication from the debugger. In this case, the data access protection violation exception does not occur.

If this exception occurs during data access with a RETI instruction in the process of an EIT sequence, the CPU stops operating and is capable of accepting a reset and break interrupt.

If this exception occurs while executing LDM0, LDM1, STM0, STM1, FLDM, or FSTM instruction, contents of execution until the occurrence are reflected in registers and memory. Check the register list (ESR:RL) for how far the instruction is executed.

Upon acceptance of the instruction access protection violation exception, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   "0" → UM "0" → S
2. The contents of the program status (PS) are saved to the system stack.
   SSP - 4 → SSP PS → (SSP)
3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.
   SSP - 4 → SSP PC → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.
   (TBR + $3E0_H$) → PC
5. A new EIT event is detected.

4.4.4 FPU Exception

An FPU exception occurs when a floating point instruction is executed. The occurrence of the floating point exception can be restrained with the floating point control register (FCR).

To prevent a subsequent instruction from being completed before detection of the FPU exception, when the FPU exception is enabled, a pipeline hazard should be generated in order to stall the pipeline. Thus the subsequent instruction will not pass the floating point instruction.

The following describes sources causing the FPU exception. For details on conditions of the occurrence, see the description of each instruction.

- When an unnormalized number has been input while the unnormalized number input is enabled.
- When the calculation result has become inexact while the inexact exception is enabled.
- When an underflow has occurred in the calculation result while the underflow exception is enabled.
- When an overflow has occurred in the calculation result while the overflow exception is enabled.
- When a division-by-zero operation has occurred while the division-by-zero exception is enabled.
- When an invalid calculation has been executed while the invalid calculation exception is enabled.

Upon acceptance of the FPU exception, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   "0" → UM "0" → S
2. The contents of the program status (PS) are saved to the system stack.
   SSP - 4 → SSP PS → (SSP)

3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.

   SSP - 4 → SSP PC → (SSP)

4. The program counter (PC) value is updated by referring to the vector table.

   (TBR + $3E8_H$) → PC

5. A new EIT event is detected.

4.4.5 Instruction Break

An instruction break generates an exception or a break based on address instructions given by the debug support unit (DSU). Upon detection of the instruction break in the user state during debugging, a break processing is carried out. Upon detection of the instruction break during normal operation, an exception processing is carried out.

The following describes the brake processing being carried out when the instruction break is accepted in the user state.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.

   "0" → UM "0" → S "4" → ILM

2. The contents of the program status (PS) are saved to the PS save register (PSSR).

   PS → PSSR

3. The contents of the program counter (PC) of an exception source instruction are saved to the PS save register (PCSR).

   PC → PCSR

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

The following describes the exception processing being carried out when the instruction break is accepted during normal operation.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and 4 is set to the interrupt level mask register (ILM).

   "0" → UM "0" → S "4" → ILM

2. The contents of the program status (PS) are saved to the system stack.

   SSP - 4 → SSP PS → (SSP)

3. The contents of the program counter (PC) of an exception source instruction are saved to the system stack.

   SSP - 4 → SSP PC → (SSP)

4. The program counter (PC) value is updated by referring to the vector table.

   (TBR + $3D4_H$) → PC

5. A new EIT event is detected.

4.4.6 Guarded Access Break

Guarded access break is a function that carries out a break processing instead of generating an exception when an instruction access protection violation or a data access protection violation occurs during debugging.

Whether each access protection violation is treated as a break or an exception processing is determined by the debugger. The guarded access break does not occur during normal operation.

If the debugger determines to carry out the break processing when instruction break has been accepted in the user state, the following operations are carried out.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.

   "0" → UM "0" → S "4" → ILM

2. The contents of the program status (PS) are saved to the PS save register (PSSR).

   PS → PSSR

3. The contents of the program counter (PC) of an exception source instruction are saved to the PS save register (PCSR).

   PC → PCSR

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

FR81 Family

4.5 Interrupts

Interrupts originate independently of the instruction sequence. They are processed by saving the necessary information to resume the currently executing instruction sequence, and then starting the processing routine corresponding to the type of the interrupt that has occurred interrupt.

Instruction loaded and executing in the CPU before the interrupt will be executed till completion. However any instruction loaded in the pipeline after the interrupt will be cancelled. Hence, after completion of the interrupt processing, processing will return to the instruction following the generation of the interrupt signal.

The following four factors cause the generation of interrupts.

- General interrupts
- Non-maskable interrupt (NMI)
- Break interrupt
- Data access error interrupt

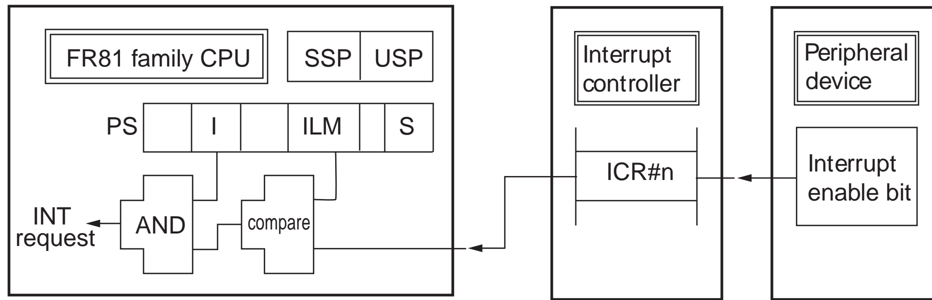
In case an interrupt is generated during the execution of stepwise division instructions, intermediate data is saved to the program status (PS) to enable resumption of processing. Therefore, if the interrupt processing program overwrites the contents of the program status (PS) data in the stack, the processor will resume the normal instruction operations following resumption of processing. However the results of the division calculation will be incorrect.

4.5.1 General interrupts

General interrupts originate as requests from in-built peripheral functions. Here, the in-built interrupt controller present in devices and external interrupt control units have been described as one of the peripheral functions.

The interrupt requests from various in-built peripheral functions are accepted via interrupt controller. There are some interrupt requests which use external interrupt control unit, taking external terminals as interrupt input terminals. Figure 4.5-1 shows the acceptance procedure of general interrupts.

Figure 4.5-1 Acceptance Procedure of General Interrupts



Each interrupt request is assigned an interrupt level by the interrupt controller, and it is possible to mask requests according to their level values. Also, it is possible to disable all interrupts by using the interrupt enable flag (I) in the condition code register (CCR).

When interrupt requests are generated by peripheral functions, they can be accepted under the following conditions.

- The level of interrupt level mask register (ILM) is higher (i.e. the numerical value is smaller) than the interrupt level set in the interrupt control register (ICR) corresponding to the vector number
- The interrupt enable flag (I) in the condition code register (CCR) is set to “1”

Interrupt control register (ICR) is a register of interrupt controller. Refer to the hardware manual of various models for details about the interrupt controller.

The following operations are performed after a general interrupt is accepted.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and the accepted interrupt request level is set to the interrupt level mask register (ILM).
   "0" → UM "0" → S Interrupt level → ILM
2. The contents of the program status (PS) are saved to the system stack.
   SSP - 4 → SSP PS → (SSP)
3. The address of the instruction next to that accepted a general interrupt is saved to the system stack.
   SSP - 4 → SSP next instruction address → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.
   (TBR + Offset) → PC
5. A new EIT event is detected.

When using general interrupts, it is required to set the interrupt level in the interrupt control register (ICR) corresponding to the vector number of the interrupt controller. Also perform the settings of the various peripheral functions and interrupt enable. Refer to the hardware manual of each model for details on interrupt controller and various peripheral functions.

4.5.2 Non-maskable Interrupts (NMI)

Non-maskable interrupts (NMI) are interrupts that cannot be masked.

Depending upon the product series, there are models which do not support NMI (there are no external NMI terminals). Refer to the hardware manual of various models to check whether NMI is supported or not.

Even if the acceptance of interrupts have been restricted by setting of "0" in the interrupt enable flag (I) of the condition code register (CCR), interrupts generated by NMI cannot be restricted. The masking of interrupt level by the interrupt level mask register (ILM) is valid. If a value above 16(10000_B) is set in the interrupt level mask register (ILM) by a program, normally "NMI" cannot be masked by the interrupt level.

The value of interrupt level mask register (ILM) is initialized to 15(01111_B) following a reset. Therefore, NMI cannot be masked until a value above 16(10000_B) by a program following a reset.

When an NMI is accepted, the following operations are performed.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and 15 is set to the interrupt level mask register (ILM).
   "0" → UM   "0" → S   "15" → ILM
2. The contents of the program status (PS) are saved to the system stack
   SSP - 4 → SSP   PS →(SSP)
3. The address of the instruction next to that accepted NMI is saved to the system stack.
   SSP - 4 → SSP   next instruction address → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.
   (TBR + $3C0_H$) → PC
5. A new EIT event is detected.

4.5.3 Break Interrupt

A break interrupt is used for break request from the debugger. The break interrupt is reported by a level, and accepted when the level is higher than that of the interrupt level mask register (ILM). The request levels from 0 to 31 are available. The level cannot be masked by the interrupt enable flag (I).

The following describes conditions to accept the break interrupt. When the conditions are met, the CPU accepts the break interrupt.

- When a break interrupt request level is higher than that of the interrupt level mask register (ILM)
- When the CPU is operating in the user state during debugging

The following describes the brake processing being carried out when the break interrupt is accepted.

1. Transition to privilege mode is carried out, the stack flag (S) cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.
   "0" → UM   "0" → S   "4" → ILM
2. The code event is determined for the instruction next to that accepted the break interrupt.

3. The contents of the program status (PS) are saved to the PS save register (PSSR).

   PS → PSSR

4. The contents of the program counter (PC) of the instruction next to that accepted the break interrupt are saved to the PC save register (PCSR).

   PC → PCSR

5. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

4.5.4 Data Access Error Interrupt

Data access error interrupts occur when a bus error occurs during data access to the buffer enabled specified area. Data access error interrupts can be enabled/disabled using the data access error interrupt enable bit (MPUCR:DEE). After a data access error interrupt occurs, a new data access error interrupt will not occur until the data access error bit (DESR:DAE) is cleared.

The data access error interrupt acceptance conditions are described below.

- The data access error interrupt enable bit (MPUCR:DEE) is enabled.
- A bus error occurs during data access to the buffer enabled specified area.

The following operations are carried out if a data access error interrupt is accepted.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.

   "0" → UM "0" → S

2. The contents of the program status (PS) are saved to the system stack.

   SSP - 4 → SSP PS → (SSP)

3. The contents of the program counter (PC) of the instruction which accepted the interrupt are saved to the system stack.

   SSP - 4 → SSP PC → (SSP)

4. The program counter (PC) value is updated.

   (TBR + $3DC_H$) → PC

5. A new EIT event is detected.

4.6 Traps

Traps are generated from within the instruction sequence. Traps are processed by first saving the necessary information to resume processing from the next instruction in the sequence, and then starting the processing routine corresponding to the type of the trap that has occurred.

Branching to the processing routine takes place after execution of the instruction that has caused the trap. The address of the instruction in which the trap occurs becomes the program counter (PC) value that is saved to the stack at the time of trap generation.

Following factors can lead to generation of traps.

- INT instruction
- INTE instruction
- Step trace traps

4.6.1 INT Instructions

The "INT #u8" instruction is used to create a trap through software. It generates a trap corresponding to the interrupt number designated in the operand.

When the INT instruction is executed, the following operations take place.

1. Transition to privilege mode is carried out, and the stack flag (S) is cleared.
   "0" → UM   "0" → S
2. The contents of the program status (PS) are saved to the system stack.
   SSP - 4 → SSP   PS → (SSP)
3. The address of the next instruction is saved to the system stack.
   SSP - 4 → SSP   next instruction address → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.
   $(TBR + 3FC_H - 4 \times u8) \rightarrow PC$
5. A new EIT event is detected.

The value of program counter (PC) saved to the system stack represents the address of the next instruction after the INT instruction.

4.6.2 INTE Instruction

The INTE instruction is used to create a software trap for debugging. A trap does not occur when the system is in the debug state during debugging, or if the step trace trap flag (SCR:T) of the program status (PS) is set. The operation of the INTE instruction varies between the user state during debugging and normal operation.

The following operations are carried out when an INTE instruction is executed during normal operation.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, and 4 is set to the interrupt level mask register (ILM).
   "0" → UM "0" → S "4" → ILM
2. The contents of the program status (PS) are saved to the system stack.
   SSP - 4 → SSP PS → (SSP)
3. The contents of the program counter (PC) of the subsequent instruction are saved to the system stack.
   SSP - 4 → SSP next instruction address → (SSP)
4. The program counter (PC) value is updated by referring to the vector table.
   (TBR + $3D8_H$) → PC
5. A new EIT event is detected.

The following operations are carried out when an INTE instruction is executed in the user state during debugging.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.
   "0" → UM "0" → S "4" → ILM
2. The contents of the program status (PS) are saved to the PS save register (PSSR).
   PS → PSSR
3. The contents of the program counter (PC) of the subsequent instruction are saved to the PS save register (PCSR).
   PC → PCSR
4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

The address saved in the system stack as program counter (PC) represents the address of the next instruction after the "INTE" instruction.

The INTE instruction should not be used within a trap processing routine of step trace trap.

4.6.3 Step Trace Traps

Step trace traps are traps used for debugging programs. Through this, a trap can be created after the execution of each instruction by setting the step trace trap flag (T) in the system condition code register (SCR). The operation of the step trace trap varies between the user state during debugging and normal operation.

A step trace trap is accepted when an instruction for which the step trace trap flag (T) is changed from "0" to "1" is executed. A step trace trap does not occur when an instruction for which the step trace trap flag (T) is changed from "1" to "0" is executed. However, for RETI instructions, a step trace trap does not occur when a RETI instruction for which the step trace trap flag (T) is changed from "0" to "1"is executed.

FR81 Family

A step trace trap is generated when the following conditions are met.

- Step trace trap flag (T) in the system condition code register (SCR) is set to "1".
- The currently executing instruction is not a delayed branching instruction
- User state in which CPU is in normal operation or debugging

Step trace trap is not generated immediately after the execution of a delayed branching instruction. It is generated after the execution of instruction within the delay slots.

When the step trace trap flag (T) is enabled, non-maskable interrupts (NMI) and general interrupts are disabled.

The following operations are carried out if a step trace trap is accepted during normal operation.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, the step trace trap flag (T) is cleared, and 4 is set to the interrupt level mask register (ILM).

   "0" → UM   "0" → S   "0" → T   "4" → ILM

2. The contents of the program status (PS) are saved to the system stack

   SSP - 4 → SSP   PS → (SSP)

3. The contents of program counter (PC) of the next instruction is saved to the system stack

   SSP - 4 → SSP   next instruction address → (SSP)

4. The program counter (PC) value is updated by referring to the vector table.

   $(TBR + 3CC_H) \rightarrow PC$

The address saved as program counter (PC) in the system stack represents the address of the next instruction after the step trace trap.

The following operations are carried out for the brake process when a step trace trap is accepted in the user state during debugging.

1. Transition to privilege mode is carried out, the stack flag (S) is cleared, the step trace trap flag (T) is cleared, 4 is set to the interrupt level mask register (ILM), and then the mode is shifted to the debug state.

   "0" → UM   "0" → S   "0" → T   "4" → ILM

2. The contents of the program status (PS) are saved to the PS save register (PSSR).

   PS → PSSR

3. The contents of the program counter (PC) of the subsequent instruction are saved to the PS save register (PCSR).

   PC → PCSR

4. An instruction is fetched from the emulator debug instruction register (EIDR1), and the handler is executed.

● Restrictions

The INTE instruction should not be used within the step trace trap handler. Use the OCD step trace function for the device installed with OCD-DSU. Do not use the step trace trap explained in this section, instead but always write "0" for step trace trap flag (T).

4.7 Multiple EIT processing and Priority Levels

When multiple EIT requests occur at the same time, priority levels are used to select one source and execute the corresponding EIT sequence.

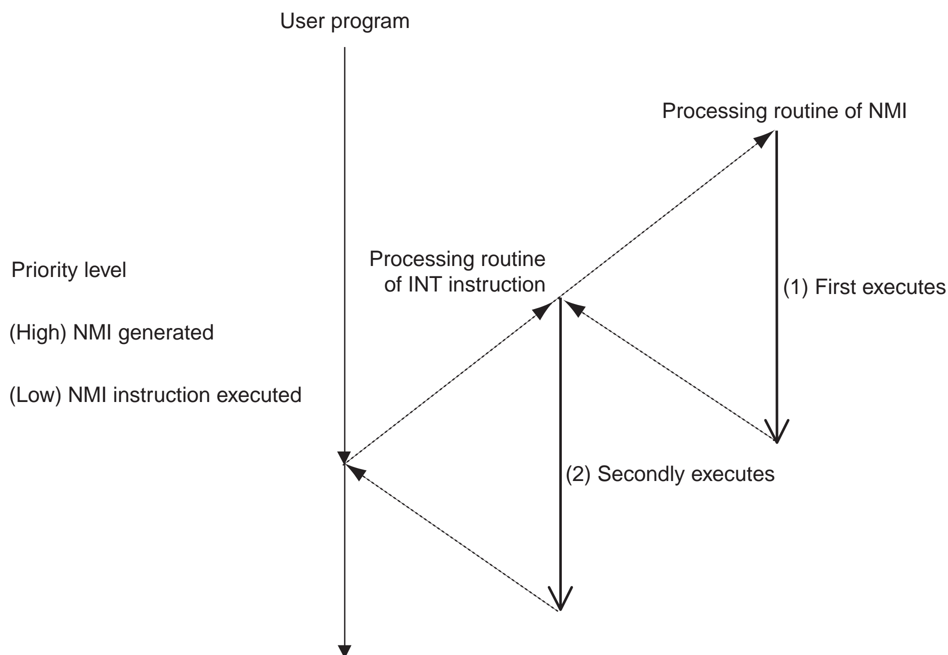
4.7.1 Multiple EIT Processing

When multiple EIT requests occur at the same time, CPU selects one source and executes the corresponding EIT sequence and then once applies EIT request detection for other sources before executing the instruction of the entry address, and this operation gets repeated.

At the time of EIT request detection, when all acceptable EIT sources have been exhausted, the CPU executes the processing routine of the EIT request accepted in the end.

When the processing is returned from the processing routine of the last EIT request accepted using the RETI instruction, the processing routine of the last but one EIT request is executed. When the processing is returned from the processing routine of the first accepted EIT request using the RETI instruction, the control returns to the user program after having processed a series of EIT processes. Figure 4.7-1 shows an example of multiple EIT processing.

Figure 4.7-1 Example of Multiple EIT Processing



For example, if A, B, C are three EIT requests that have occurred simultaneously, and have been accepted in the order of B, C, A, the execution of the processing routine will be in the order A, C, B.

4.7.2 Priority Levels of EIT Requests

The sequence of accepting each request and executing the corresponding processing routines when multiple EIT request occur simultaneously, is decided by two factors - by the priority levels of EIT requests, and, how other EIT requests are to be masked when one EIT request has been accepted.

At the time when an EIT request occurs, and at the time of completion of an EIT sequence, the detection of EIT requests being generated at that time is performed, and which EIT request will be accepted will be decided. At the time of completion of the EIT sequence, the detection of EIT requests is carried out under the condition where masking has been done for the EIT sources other than the EIT request accepted just a while back. Table 4.7-1 shows the priority levels of EIT requests and masking of other sources.

Table 4.7-1 Priority Levels of EIT Requests & Masking of Other Sources

Priority level	EIT Source	Masking of other sources	ILM after updated
1	Reset	Other sources discarded	15
2	Instruction break Guarded access break	All factors given lower priority	4
3	Invalid instruction exception Instruction access protection exception Data access protection exception FPU exception	All factors given lower priority	-
4	INT instruction	I flag = 0	
5	INTE instruction	All factors given lower priority	4
6	General interrupt	ILM= level of source accepted	ICR
7	NMI	ILM=15	15
8	Data access error interrupt	-	-
9	Break interrupt	All factors given lower priority	Request level
10	Step Trace Traps	All factors given lower priority	4

There are times when the value of interrupt level mask register (ILM) gets modified due to the EIT request accepted earlier and the other EIT sources occurring simultaneously get masked and cannot be accepted. In such a case, until the processing routine of EIT sources that have occurred simultaneously have been executed and the control has returned to the user program, the user interrupt is suspended and is re-detected at the time of resumption of the user program.

4.7.3 EIT Acceptance when Branching Instruction is Executed

No interrupts are accepted when a branching instruction is executed for delayed branching instruction. Also, when an exception occurs in the delay slot, branching is cancelled, and the program counter (PC) for branching instruction is saved. Interrupts and traps are accepted for delay slot instruction. Table 4.7-2 shows the EIT acceptance and saved PC value for branching instructions.

Table 4.7-2 EIT acceptance and saved PC value for branching instruction

EIT acceptance instruction		Branching instruction				Delay slot instruction			
EIT type		Exception		Interrupt/trap		Exception		Interrupt/trap	
Branching	Delay slot	Acceptance	Saved PC value	Acceptance	Saved PC value	Acceptance	Saved PC value	Acceptance	Saved PC value
Yes	None	❍	PC	❍	Branching destination	-	-	-	-
	Yes	❍	PC	×	-	❍	Branching instruction	❍	Branching destination
No	None	❍	PC	❍	Subsequent instruction	-	-	-	-
	Yes	❍	PC	×	-	❍	PC	❍	Subsequent instruction

FR81 Family

4.8 Timing When Register Settings Are Reflected

The timing when the new values are reflected after the interrupt enable flag (I) of program status (PS) and the value of interrupt level mask register (ILM) are modified will be explained in this section.

4.8.1 Timing when the interrupt enable flag (I) is requested

The interrupt request (enable/disable) is reflected from the instruction which modifies the value of interrupt enable flag (I).

Figure 4.8-1 shows the timing of reflection of the interrupt enable flag (I) when interrupt enable is set to (I=1), and Figure 4.8-2 shows the timing of reflection of the interrupt enable flag (I) when interrupt disable is set to (I=0).

Figure 4.8-1 Timing of reflection of interrupt enable flag (I) when interrupt enable is set to (I=1)

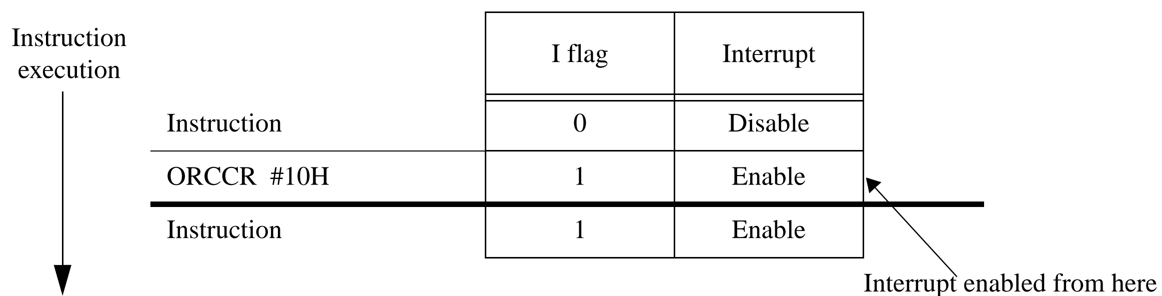
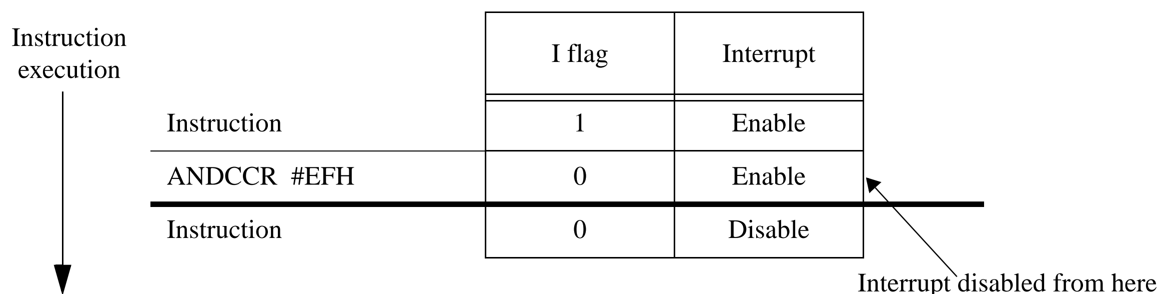


Figure 4.8-2 Timing of reflection of interrupt enable flag (I) when interrupt disable is set to (I=0)

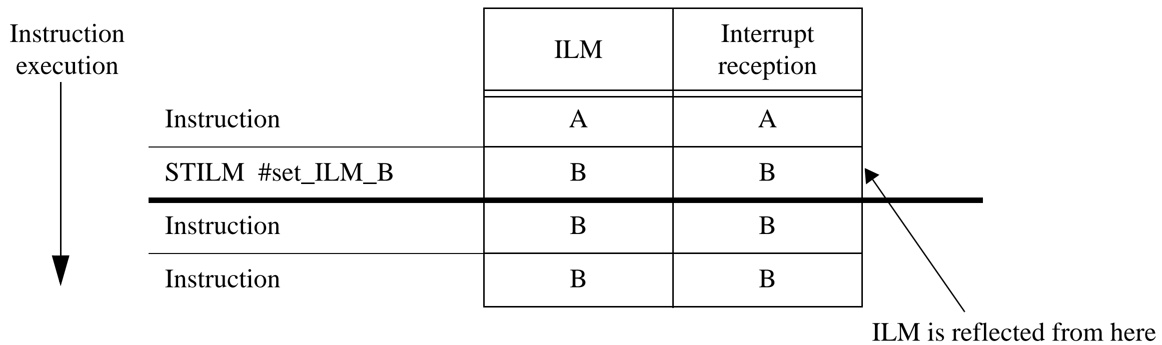


4.8.2 Timing of Reflection of Interrupt Level Mask Register (ILM)

Acceptance to interrupt request is reflected from the instruction which modifies the value of interrupt level mask register (ILM).

Figure 4.8-3 shows the timing of reflection when the interrupt level mask register (ILM) is modified.

Figure 4.8-3 Timing of reflection when the Interrupt level mask register (ILM) is modified



"set_ILM_B" is a value of the interrupt level mask register (ILM) to be newly assigned. As in the case of STILM #30, assign a numeric value of 0 to 31.

4.9 Usage Sequence of General Interrupts

General interrupts accept interrupt requests from in-built peripheral functions and external terminals, and perform EIT processing. The general points of caution of programming while using general interrupts have been described here. Refer to the hardware manual of various models as the detailed procedure differs as per the peripheral function.

4.9.1 Preparation while using general interrupts

Before using general interrupts, settings for EIT processing need to be made. Perform the following settings in the program beforehand.

- Set values in the vector table (defined as data)
- Set up the system stack pointer (SSP) values
- Set up the table base register (TBR) value as the initial address in the vector table
- Set a value of above 16(10000_B) in the interrupt level mask register (ILM)
- Set the value of "1" in the interrupt enable flag (I)

After the above settings, the settings of the peripheral functions are performed. In case of peripheral functions which use general interrupts, two bits in the register of the peripheral functions require to be set - a flag bit that indicates that a phenomenon which can become an interrupt source has occurred, and an interrupt enable bit which uses this flag bit to enable or disable the interrupt request.

The peripheral function verifies the operation halt status, the disable of interrupt request, and that the flag bit has been cleared. This state is achieved following a reset.

In case the peripheral function is engaged in some operation, the interrupt request is disabled and the flag bit cleared after the operation of the peripheral function has been halted.

The interrupt level is set in the interrupt control register (ICR) of the interrupt controller. As multiple interrupt control registers (ICR) are available corresponding to various vector numbers in the, please set an interrupt control register (ICR) corresponding to the vector number of the interrupt begin used.

The operation of the peripheral function is resumed after clearing the flag bit and enabling the interrupt request.

4.9.2 Processing during an Interrupt Processing Routine

After the interrupt request for a general interrupt has been accepted in the CPU as EIT following its generation, the control moves to the interrupt processing routine after the execution of the EIT sequence.

Vector numbers are assigned to each source of general interrupts, and the interrupt processing routine corresponding to these vector numbers are started. The interrupt sources and vector numbers do not necessarily have a one-to-one correspondence, and at times the same vector number is assigned to multiple interrupt sources. In such a case, the same interrupt processing routine is used for multiple interrupt sources.

Right in the beginning of the interrupt processing routine, the flag bit which indicates an interrupt source is verified. If the flag bit has been set, interrupt request for that interrupt is generated and the required processing (program) is executed after clearing the flag bit. In case, the same vector offset is being used for multiple interrupt sources, there are multiple flag bits indicating interrupt sources, and each of them are identified and processed in the same manner.

It is necessary to clear the flag bit while the interrupt of that particular interrupt source is in the disabled state. When the interrupt processing routine is started after the execution of the EIT sequence, the interrupt level of the general interrupt is stored in the interrupt level mask register (ILM) and the general interrupt of that interrupt level is disabled. Make sure to clear the flag bit at the end of the interrupt processing without modifying the interrupt level mask register (ILM).

The control is returned from the interrupt processing routine by the RETI instruction.

4.9.3 Points of Caution while using General Interrupts

Interrupt requests are enabled either when the corresponding flag bit has been cleared, or at the time of clearing the flag bit. Enabling interrupt requests when the flag bit is in the set state, leads to the generation of interrupt request immediately.

While enabling interrupt requests, do not clear flag bit besides the interrupt processing routine. Flag bit should be cleared at the time of disabling interrupt request.

In case a flag bit is cleared when a peripheral function is performing an operation, there are times when the flag bit cannot be cleared if the clearing of flag bit by writing to the register and the occurrence of a phenomenon which can be an interrupt source take place simultaneously or at a very close interval. Whether a flag bit will be cleared or not when the clearing of flag bit and the occurrence of a phenomenon that can become an interrupt source take place simultaneously, differs from one peripheral function to the other.

4.10 Precautions

The precautions of The reset and the EIT processing described here.

4.10.1 Exceptions in EIT Sequence and RETI Sequence

If a data access protection violation exception (including guarded access break) or invalid instruction exception (data access error) occurs in the EIT or RETI sequence, since access to the system stack area is disabled, the CPU goes into the inactive state. To restore the system from this state, reset the system or execute a break interrupt from the debugger.

At this time, the data access protection violation exception or invalid instruction exception (data access error) cannot be accepted, and the processing is stopped immediately. The reset process/break process starts when a reset or break request is detected in the CPU stopped. If the CPU shifts to this stop state, since the EIT or RETI sequence is stopped in the middle of execution, it is impossible to execute the user program with this condition.

4.10.2 Exceptions in Multiple Load and Multiple Store Instructions

If a data access protection violation exception, invalid instruction exception (data access error), or guarded access break (data access) occurs when the LDM0, LDM1, STM0, STM1, FLDM or FSTM instruction is executed, the result of the processing up to this point is reflected in the memory or R15 (SSP/USP). The list of registers that have not been executed is stored in the register list of the exception status register (ESR).

4.10.3 Exceptions in Direct Address Transfer Instruction

If a data access protection violation exception, invalid instruction exception (data access error), or guarded access break (data access) occurs when data is transferred from the direct area to the memory by the direct address transfer instruction (DMOV), the I/O register value is updated when the I/O register value in the direct area is changed by reading.

CHAPTER 5
PIPELINE OPERATION

This chapter explains the chief characteristics of FR81 family CPU like pipeline operation, delayed branching processing etc.

5.1 Instruction execution based on Pipeline

5.2 Pipeline Operation and Interrupt Processing

5.3 Pipeline hazards

5.4 Non-block loading

5.5 Delayed branching processing

5.1 Instruction execution based on Pipeline

FR81 Family CPU processes a instruction using a pipeline operation. This makes it possible to process to process nearly all instructions in one cycle. FR81 Family has two pipelines: an integer pipeline and floating point pipeline.

Pipeline operation divides each type of step that carries out interpretation and execution of instructions of CPU in to stages, and simultaneously executes different stages of each instruction. Instruction execution that requires multiple cycles in other processing methods is apparently conducted in one cycle here.

Processing of both the integer pipeline and floating point pipeline are common up to the decoding stage, and independent processing is carried out for each pipeline from the execution and subsequent stages. The process sequence for each pipeline differs from the sequence of issuing instructions. However, the processing result that has been acquired by following the program sequence procedure is guaranteed.

5.1.1 Integer Pipeline

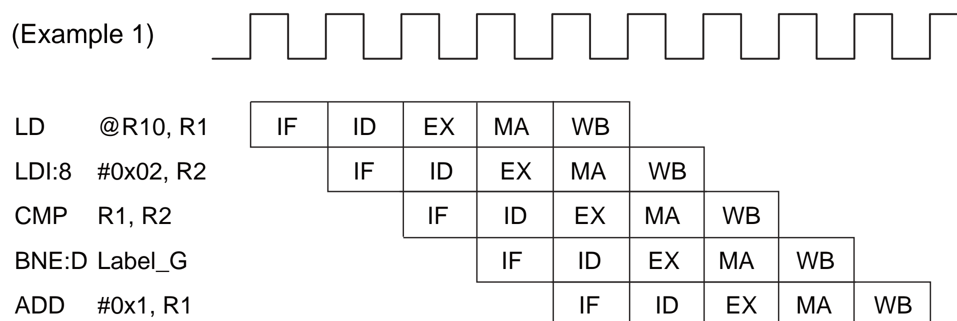
Integer pipeline is a 5-stage pipeline compatible with FR family. A 4-stage load buffer is provided for non-blocking loading.

The integer pipeline has the following 5-stage configuration.

- IF Stage: Fetch Instruction
  Instruction address is generated and instruction is fetched.
- ID Stage: Decode Instruction
  Fetched instruction is decoded. Register reading is also carried out.
- EX Stage: Execute Instruction
  Computation is executed.
- MA Stage: Memory Access
  Loading or access to storage is executed against the memory.
- WB Stage: Write Back to register
  Computation result (or loaded memory data) is written in the register.

Example of the integer pipeline operation (1) is shown in Figure 5.1-1 and Example of integer pipeline operation (2) is shown in Figure 5.1-2.

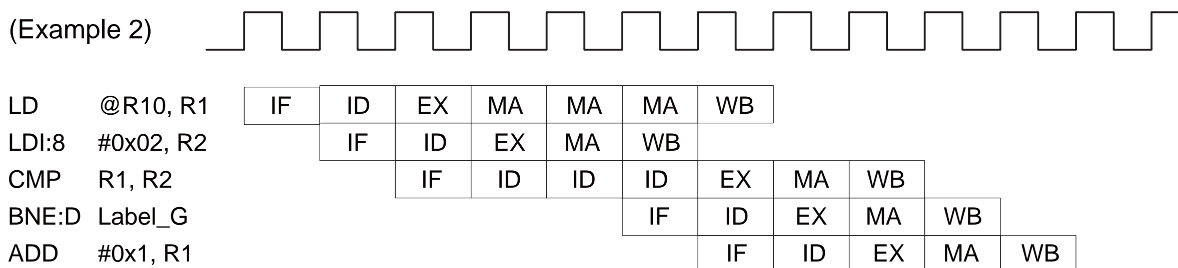
Figure 5.1-1 Example of integer pipeline operation (1)



In principle, execution of instructions is carried out at one instruction per cycle. However, multiple cycles are necessary for the execution of instruction in case of load store instruction accompanied by memory wait, non-delayed branching instruction, and multiple cycle instruction. The speed of instruction execution is also reduced in cases where there is a delay in the supply of instructions, such as internal conflict of bus in the CPU, instruction execution through external bus interface etc.

Normally, instructions are executed sequentially in the integer pipeline. For example, if instruction A enters the pipeline before instruction B, it invariably reaches WB stage before instruction B. However, when the register used in Load instruction (LD instruction) is not used in the subsequent instruction, the subsequent instruction is executed before the completion of execution of load instruction based on the non-blocking loading buffer.

Figure 5.1-2 Example of integer pipeline operation (2)



MA stage is prolonged in case of Load instruction (LD instruction) till the completion of reading of the loaded data. However, the subsequent instruction is executed as it is, if the register used in the load instruction is not used in the subsequent instruction.

In the Example given in Figure 5.1-1, loading is carried out in R1 (load value is written in R1) based on preceding LD instruction, and R1 contents are referred to in the subsequent CMP instruction. Since the loaded data returns in 1 cycle, execution of instructions is sequential.

Similarly in the Example given in Figure 5.1-2, R1 that writes load value with LD instruction is used in the CMP instruction. Since the loaded data does not return in 1 cycle, execution till LDI:8 instruction is carried out and CMP instruction is made to wait at the ID stage by register hazard.

5.1.2 Floating Point Pipeline

The floating point pipeline is a 6-stage pipeline used to execute floating point calculations. The IF stage and ID stage are common with the integer pipeline.

The floating point pipeline has the following 5-stage configuration.

- IF Stage: Fetch Instruction
  Instruction address is generated and instruction is fetched.
- ID Stage: Decode Instruction
  Fetched instruction is decoded. Register reading is also carried out.
- E1 Stage: Execute Instruction 1
  Computation is executed. Multiple cycles may be required depending on the instruction.
- E1 Stage: Execute Instruction 2
  The result is rounded and normalized.
- WB Stage: Write Back to register
  Computation result is written in the register.

5.2 Pipeline Operation and Interrupt Processing

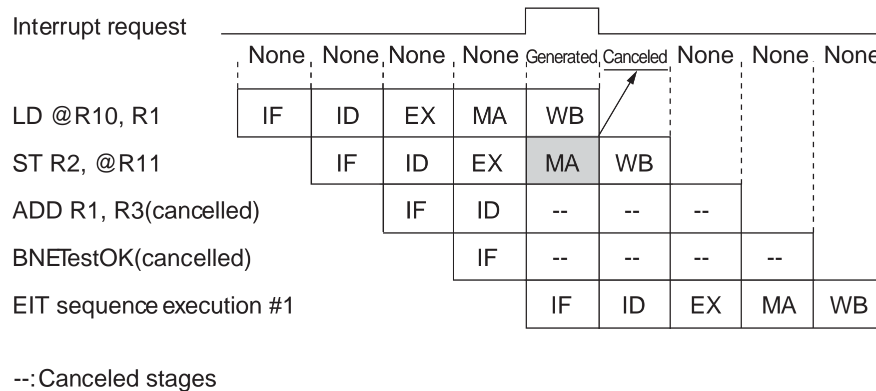
It is possible at times that an event wherein it appears that interrupt request is lost after acceptance of interrupt, if the flag that causes interrupt in the interrupt-enabled condition, because pipeline operation is conducted, occurs.

5.2.1 Mismatch in Acceptance and Cancellation of Interrupt

Because CPU is carrying out pipeline processing, pipeline processing of multiple instructions is already executed at the time of acceptance of interrupt. Therefore, in case corresponding interrupt cancellation processing among the instructions under execution in the pipeline (For example, clearing of flag bits that cause interrupt) is carried out, branching to corresponding interrupt processing program is carried out normally but when control is transferred to interrupt processing, the interrupt request is at times already over (Flag bits that cause interrupt having been cleared).

An Example of Mismatch in Acceptance and cancellation of interrupt is shown is Figure 5.2-1.

Figure 5.2-1 Example of Mismatch in Acceptance and Cancellation of interrupt



This type of phenomenon does not occur in case of exceptions and trap, because the operation for request cancellation cannot be carried out in the program.

5.2.2 Method of preventing the mismatched pipeline conditions

Mismatch in Acceptance and Deletion of interrupt can occur in case flag bits that cause interrupt are cleared while interrupt request is enabled in the peripheral functions.

To avoid such a phenomenon, programmers should set the interrupt enable flag (I) at "0", disable interrupt acceptance in CPU and clear the flag bits that cause an interrupt.

5.3 Pipeline hazards

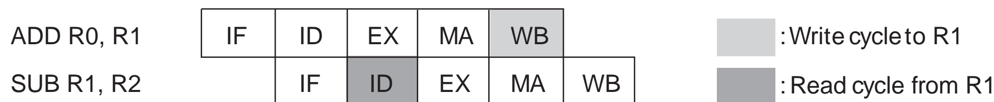
The FR81 Family CPU executes program steps in the order in which they are written and is therefore equipped with a function that detects the occurrence of data hazards and construction hazards, and stops pipeline processing when necessary.

5.3.1 Occurrence of data hazard

A data hazard occurs if dependency that refers or updates the register exists in between the preceding and subsequent instructions. The CPU may simultaneously process one instruction that involves writing values to a register, and a subsequent instruction that attempts to refer to the same register before the write process is completed.

An example of a data hazard is shown in Figure 5.3-1. In this case, the reading of R1 used as the address will read the value before the modification, as the read timing precedes the writing to R1 requested by the just previous instruction. (Actually, the data hazard is avoided, and the modified value is read.)

Figure 5.3-1 Example of a data hazard

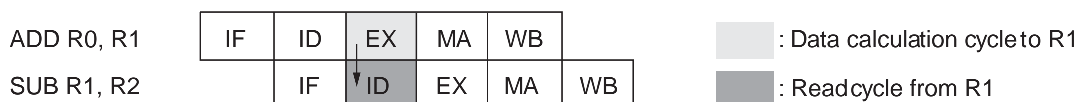


5.3.2 Register Bypassing

Even when a data hazard does occur, it is possible to process instructions without operating delays if the data intended for the register to be accessed can be extricated from the preceding instruction. This type of data transfer processing is called register bypassing.

An example of Register Bypassing is indicated in Figure 5.3-2. In this example, instead of reading the R1 in the ID stage of SUB instruction, the program uses the results of the calculation from ADD instruction (before the results are written to the register) and thus executes the instruction without delay.

Figure 5.3-2 Example of a register bypass

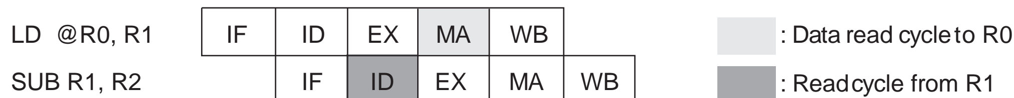


5.3.3 Interlocking

Instructions that are relatively slow in loading data to the CPU may cause data hazards that cannot be handled by register bypassing.

In the example Figure 5.3-3, data required for the ID stage of the SUB instruction must be loaded to the CPU in the MA stage of the LD instruction, creating a data hazard that cannot be avoided by the bypass function.

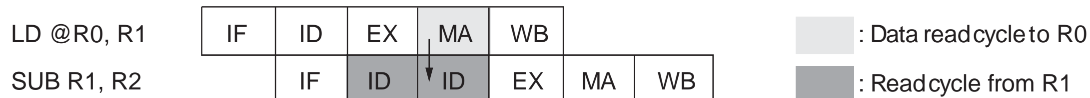
Figure 5.3-3 Example: Data Hazard that cannot be avoided by Bypassing



In cases such as this, the CPU executes the instruction correctly by pausing before the execution of subsequent instruction. This function is called interlocking.

In the example in Figure 5.3-4, the ID stage of the SUB instruction is delayed until the data is loaded from the MA stage of the LD instruction.

Figure 5.3-4 Example of Interlocking



5.3.4 Interlocking produced by reference to R15 after Changing the Stack flag (S)

The general purpose register R15 is designed to function as either the system stack pointer (SSP) or user stack pointer (USP). For this reason the FR Family CPU is designed to automatically generate an interlock whenever a change to the stack flag (S) in the program status (PS) is followed immediately by an instruction that references the R15. This interlock enables the CPU to reference the SSP or USP values in the order in which they are written in the program.

Hardware design similarly generates an interlock whenever a TYPE-A format instruction immediately follows an instruction that changes the value of Stack flag (S). For information on instruction formats, see Section "6.2.3 Instruction Formats".

5.3.5 Structural Hazard

A structural hazard occurs if a resource conflict occurs between instructions which use the same hardware resource. If this hazard is detected, the pipeline is interlocked to pause the processing of subsequent instruction until the hazard is eliminated.

5.3.6 Control Hazard

A control hazard occurs when the next instruction cannot be fetched before the branching instruction is complete. In FR81 Family, to reduce penalty due to this control hazard, the pre-fetch function that bypasses the branch destination address from the ID stage and the delayed branching instruction have been added. Therefore, penalties do not become apparent.

● Pre-fetch function

FR81 Family CPU has a 32-bit 4-stage pre-fetch buffer, and fetches a subsequent instruction of consecutive addresses as long as the buffer is not full. However, when a branching instruction is decoded, the instruction is fetched from the branching destination regardless of the condition. If an instruction is branched, the instruction in the pre-fetch buffer is discarded, and the subsequent instruction in the branching destination will be pre-fetched. If not branched, the instruction in the branching destination is discarded, and the instruction in the pre-fetch buffer will be used.

● Delayed branching processing

Delayed branching processing is the function to execute the instruction immediately following the branching instruction for pipeline operation by one, regardless of whether the branching is successful or unsuccessful. The position immediately following a branching instruction is called the delay slot. Instructions that can be placed in the delay slot should be executable in one state having 16-bit length. Placing an instruction that does not fit in the delay slot will result an invalid instruction exception to occur. Refer to Appendix A.3 for the list of instructions that can be placed in delay slot.

FR81 Family

5.4 Non-block loading

Non-block loading is carried out in FR81 Family CPU. A maximum of 4 loading instructions can be issued with precedence.

In non-block loading, the subsequent instruction is executed without waiting for the completion of loading instruction, if the general-purpose register in which the load instruction value is stored is not referred in the subsequent instruction.

As shown below, when register R1 that stores data value based on LD instruction is referred to in the subsequent ADD instruction, the ADD instruction is executed after storing R1 value based on LD instruction.

```
LD      @10,R1
ADD     R1,R2     ; waits for completion of execution of preceding LD instruction
```

As shown below, ADD instruction is executed without waiting for the completion of execution of LD instruction when R1 that stores data value by LD instruction is not referred to in the subsequent ADD instruction. After that, at the time of execution of SUB instruction that references R1, if the preceding LD instruction is not already executed, the SUB instruction is executed after waiting for the completion of execution of that LD instruction.

```
LD      @10,R1
ADD     R2,R3     ; Does not wait for completion of execution of preceding LD instruction
SUB     R1,R3     ; waits for completion of execution of preceding LD instruction
```

A maximum of 4 load instructions can be executed with precedence. It can also be used in the following way for issuing multiple LD instructions with precedence.

```
LD      @100,R1   ; LD instruction (1)
LD      @104,R2
LD      @108,R3
LD      @112,R4   ; a maximum of four LD instructions can be issued with precedence
ADD     R5,R6     ; executed without waiting for the completion of execution of preceding LD
                    instruction
SUB     R6,R0
ADD     R1,R5     ; executed after completion of execution of preceding LD instruction (1)
```

5.5 Delayed branching processing

Because FR81 Family CPU features pipeline operation, the loading of the instruction is already completed at the time of execution of branching instruction. The processing speeds can be improved by using the delayed branching processing.

5.5.1 Example of branching with non-delayed branching instructions

Non-delayed branching instruction executes instructions in the order of program but the execution speed drops down by 1 cycle as compared to delayed branching instruction when branching.

In a pipeline operation, by the time the CPU recognizes an instruction as a branching instruction the next instruction has already been loaded. To process the program as written, the instruction following the branching instruction must be cancelled in the middle of execution. Branching instructions that are handled in this manner are non-delayed branching instructions.

The example of processing non-delayed branching instruction with fulfilled branching conditions is given in Figure 5.5-1 which shows that execution of the "ST R2,@R12" instruction (instruction placed immediately after branching instruction) that had started pipeline operation before fetching instruction from the branching destination is cancelled in the middle. Due to this, program processing happens as the program is written, but branching instruction apparently takes 2 cycles for completion.

Figure 5.5-1 Example of processing of Non-Delayed Branching instruction (Branching conditions satisfied)

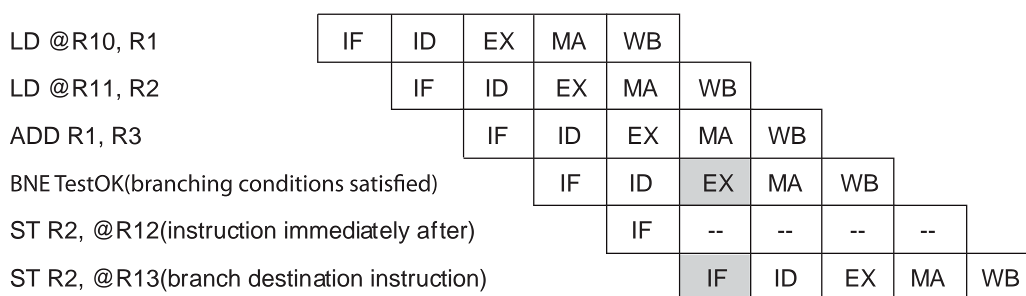
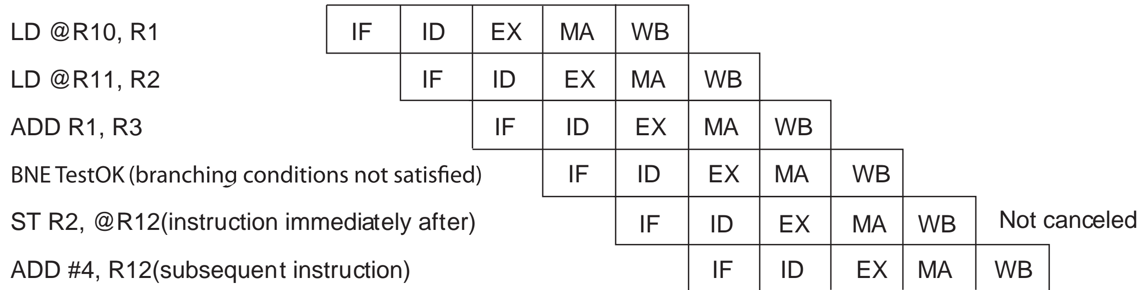


Figure 5.5-2 shows an example of processing a non-delayed branching instruction when branching conditions are not fulfilled. In this example, the "ST R2,@R12" instruction (instruction kept immediately after branching instruction) that started pipeline processing before fetching instruction from the branching destination is executed without being cancelled. The processing of program happens as written in the program since the instructions are executed sequentially, without branching, and branching instruction execution speed is apparently of 1 cycle.

FR81 Family

Figure 5.5-2 Example of processing of Non-Delayed Branching instruction (Branching conditions not satisfied)



5.5.2 Example of processing of delayed branching instruction

Delayed branching instructions are processed with an apparent execution speed of 1 cycle, regardless of whether or not branching conditions are satisfied. When branching occurs, this is one cycle faster than using non-delayed branching instructions. However, the apparent order of instruction processing is inverted in cases where branching occurs.

An instruction immediately following a branching instruction will already be loaded by the CPU by the time the branching instruction is executed. This position is called the delay slot. A delayed branching instruction is a branching instruction that executes the instruction in the delay slot regardless of whether or not branching conditions are satisfied.

Figure 5.5-3 shows an example of processing a delayed branching instruction when branching conditions are satisfied. In this example, the branch destination instruction "ST R2,@R13" is executed after the instruction "ST R2,@R12" in the delay slot. As a result, the branching instruction has an apparent execution speed of 1 cycle. However, the instruction "ST R2,@R12" in the delay slot is executed before the branch destination instruction "ST R2,@R13" and therefore the apparent order of processing is inverted.

Figure 5.5-3 Example of processing of Delayed Branching instruction (Branching conditions satisfied)

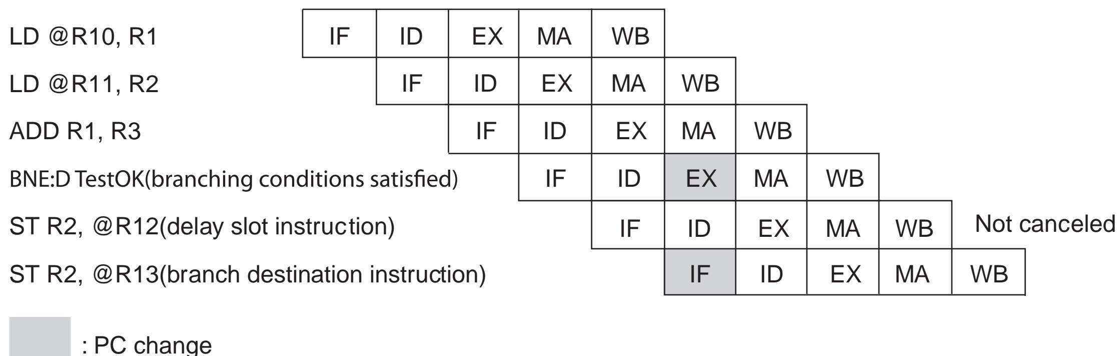
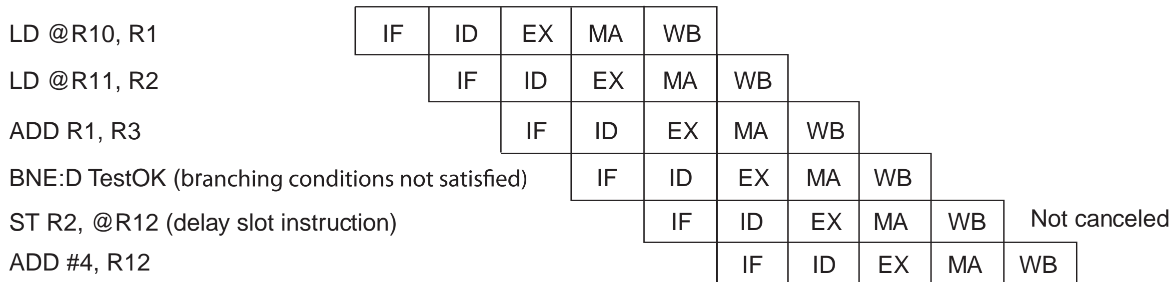


Figure 5.5-4 shows an example of processing a delayed branching instruction when branching conditions are not satisfied. In this example, the instruction "ST R2,@R12" in delay slot is executed without being cancelled. As a result, the program is processed in the order in which it is written. The branching instruction requires an apparent processing time of 1 cycle.

Figure 5.5-4 Example of processing of Delayed Branching instruction (Branching conditions not satisfied)



CHAPTER 6

INSTRUCTION OVERVIEW

This chapter presents an overview of the instructions used with the FR81 Family CPU.

6.1 Instruction System
6.2 Instructions Formats
6.3 Data Format
6.4 Read-Modify-Write type Instructions
6.5 Branching Instructions and Delay Slot
6.6 Step Division Instructions

6.1 Instruction System

FR81 Family CPU has the integer type instruction of upward compatibility with FR80 Family and floating point type instruction executed by FPU.

6.1.1 Integer Type Instructions

Integer type instructions, in addition to instruction type of general RISC CPU, is also compatible with logical operation optimized for embedded use, bit operation and direct addressing instructions.

Integer type instructions of FR81 Family CPU can be divided into the following 15 groups.

- Add/Subtract Instructions

These are the Instructions to carry out addition and subtraction between general-purpose registers or a general-purpose register and immediate data. They also enable computation with carry used in multi-word long computation or computations where flag value of Condition Code Register (CCR) convenient for address calculation is not changed.

- Compare Instructions

These are the Instructions to carry out subtraction between general-purpose registers or a general-purpose register and immediate data and reflect the results in the flag of Condition Code Register (CCR).

- Logical Calculation Instructions

These are the Instructions to carry out logical calculation for each bit between general-purpose registers or a general-purpose register and memory (including I/O). Logical calculation types are logical product (AND), logical sum (OR), and exclusive logical sum (EXOR). Memory addressing is register indirect.

- Bit Operation Instructions

These are the Instructions to carry out logical calculation between memory (including I/O) and immediate value and operate directly for each bit. Logical calculation types are logical product (AND), logical sum (OR), and exclusive logical sum (EXOR). Memory addressing is register indirect.

- Multiply/Divide Instructions

These are the instructions to carry out multiplication and division between general-purpose register and multiplication/division result register. There are 32 bit × 32 bit, 16 bit × 16 bit multiplication instructions and step division instructions to carry out 32 bit ÷ 32 bit division.

- Shift Instructions

These are the instructions to carry out shift (logical shift, arithmetic shift) of general-purpose registers. By specifying general-purpose register or immediate data, shift (Barrel Shift) of multiple bits can be specified at once.

● Immediate Data Transfer Instructions

These are the instructions to transfer immediate data to general-purpose registers and can transfer immediate data of 8bit, 20 bit, and 32 bit.

● Memory Load Instructions

These are the instructions to load from memory (including I/O) to general-purpose registers or dedicated registers. They can transfer data length of 3 types namely, bytes, half-words and words and memory addressing is register indirect.

During memory addressing of some Instructions, Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

● Memory Store Instructions

These are the instructions to store from general-purpose register or dedicated register to memory (Including I/O). They can transfer data length of 3 types namely, bytes, half-words and words and memory addressing is register indirect.

During memory addressing of some Instructions, Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

● Inter-register Transfer Instructions/Dedicated Register Transfer Instructions

These are the instructions to transfer data between general-purpose registers or a general-purpose register and dedicated register.

● Non-delayed Branching Instructions

These are the instructions that do not have delay slot and carry out branching, sub-routine call, interrupt and return.

● Delayed Branching Instructions

These are the instructions that have delay slot and carry out branching, sub-routine call, interrupt and return. Delay slot instructions are executed when branching.

● Direct Addressing Instructions

These are the instructions to transfer data between general-purpose register and memory (INCLUDING I/O) or between two memories. Addressing is not register indirect but direct specification with operand of instruction.

In some instructions, in combination with specific general-purpose registers, access is made in combination with increment/decrement Register Indirect addressing.

● Bit Search Instructions

These are the instructions that have been added to FR81/FR80 Family CPU. They search 32-bit data of general-purpose register from MSB and obtain the first "1" bit, "0" bit and bit position of change point (distance of bit from MSB).

They correspond to bit search module packaged in the family prior to FR81/FR80 Family (FR30 Family, FR60 Family etc.) as peripheral function.

● Other Instructions

These are the instructions to carry out flag setting, stack operation, sign/zero extension etc. of Program Status (PS). There are also high-level language compatible Enter Function/Leave Function, Register Multi load/store Instructions.

See “A.2 Instruction Lists” to know about the groups and types of Instructions.

6.1.2 Floating Point Type Instructions

The floating point type instructions is an instruction added in FR81 family CPU. The floating point type instructions is divided into the following six groups.

● FPU Memory Load Instruction

This is an instruction to load data from memory to the floating point register. Memory addressing is register indirect. Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

● FPU Memory Store Instruction

This is an instruction to store data from the floating point register in memory. Memory addressing is register indirect. Displacement Register Indirect or Increment/Decrement Register Indirect Address is possible.

● FPU Single-Precision Floating Point Calculation Instruction

This is an instruction to perform single-precision floating point calculations.

● FPU Inter-Register Transfer Instruction

This is an instruction to transfer data between floating point registers or between the floating point register and general-purpose register.

● FPU Branching Instruction without Delay

This is a conditional branching instruction without a delay slot.

● FPU Branching Instruction with Delay

This is a conditional branching instruction with a delay slot.

FR81 Family

6.2 Instructions Formats

This part describes about Instruction Formats of FR81 Family CPU.

6.2.1 Instructions Notation Formats

● Integer type instruction

The integer type instruction is 2 operand format. There are 3 types of Instruction notation formats depending on the number of operands. Instruction notation formats are as follows.

<Mnemonic> <Operand 1> <Operand 2>

Mnemonic calculations are carried out between operand 2 and operand 1 and the results are stored at operand 2.

Ex: ADD R1,R2 ; R2 + R1 -> R2

<Mnemonic> <Operand 1>

Operations are designated by a mnemonic and use operand 1.

Ex: JMP @R1 ; R1 -> PC

<Mnemonic>

Operations are designated by a mnemonic.

Ex: NOP ; No Operation

Operands have general-purpose register, dedicated register, immediate data and combinations of part of general-purpose register and immediate data. Operand format varies depending on Instruction.

● Floating point type instruction

Floating point type instruction is 3 operand format. The following description formats are added.

<Mnemonic> <Operand 1> <Operand 2> <Operand 3>

Mnemonic calculations are executed between operand 1 and operand 2 and the results are stored in operand 3. For some of the instructions, calculations are executed between operand 3 and the calculation result of operand 1 and operand 2, and then the results are stored in operand 3.

Ex: FADDs FR1, FR2, FR3 ; FR1 + FR2 -> FR3

6.2.2 Addressing Formats

There are several methods for address specification when accessing memory in the memory space or I/O register. Addressing format varies depending on Instruction.

@General-purpose Registers

It is Register Indirect Addressing. Address indicated by the content of the general-purpose register is accessed.

@(R13, General-purpose Register)

Address where virtual accumulator (R13) and contents of general-purpose register are added is accessed.

@(R14,Immediate Data)

Address where contents of Frame Pointer (R14) and immediate data are added is accessed. Immediate data is specified in the multiples of data size (word, half word, byte).

@(R15, Immediate Data)

Address where contents of Stack Pointer (R15) and immediate data are added is accessed. Immediate data is specified in the multiples of data size (word, half word, byte).

@R15+

Write access to the address indicated by the contents of Stack Pointer (R15) is made. 4 will be added to the stack pointer (R15).

@-R15

Read access to the address which is deduction of 4 from the contents of Stack Pointer (R15) is made. 4 will be deducted from the Stack Pointer (R15).

@ Immediate Data

It is direct addressing. Address indicated by immediate data is accessed.

@R13+

Access to address indicated by the contents of virtual accumulator (R13) is made. Data size (Bytes) will get added to virtual accumulator (R13).

@(BP, Immediate Data)

Address where the base pointer (BP) and immediate data are added is accessed. Immediate data is specified in the multiples of data size (word, half word, byte).

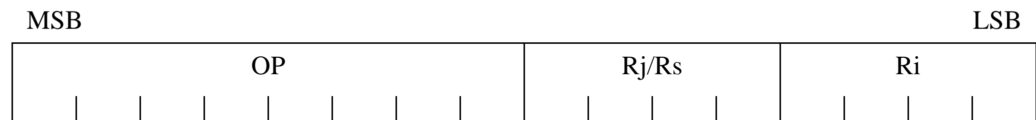
6.2.3 Instruction Formats

FR81 Family CPU Instructions are 16-bit in length. Bit configuration of Instructions varies depending on configuration of operands of Instructions. Bit configuration of Instructions classified into groups is called Instruction Formats.

There are 14 types of Instructions Formats TYPE-A through TYPE-N.

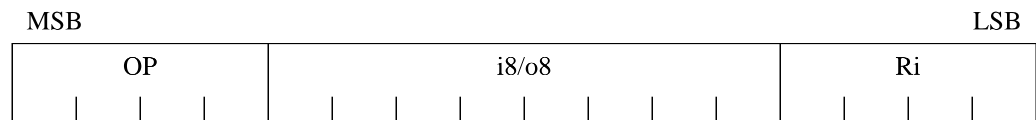
TYPE-A

It has 8-bit OP Code (OP) and two Register designated fields (Rj/Rs,Ri)



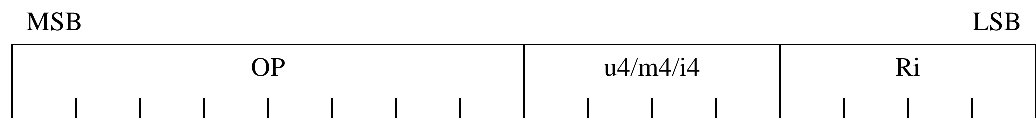
TYPE-B

It has 4-bit OP Code (OP) and 8-bit immediate data fields (i8/o8), register designated field (Ri)



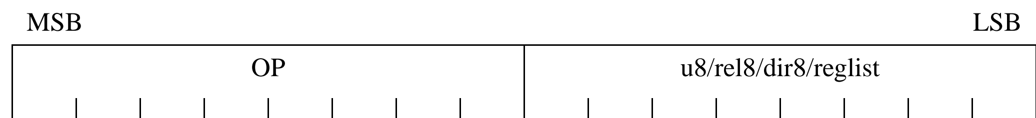
TYPE-C

It has 8-bit OP Code (OP) and 4-bit immediate data fields (u4/m4/i4), register designated field (Ri)



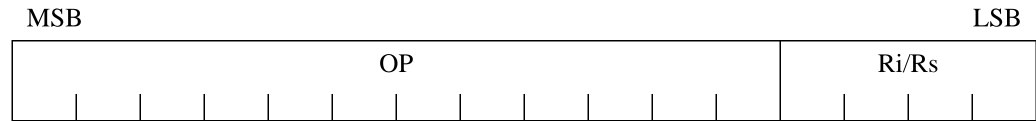
TYPE-D

It has 8-bit OP Code (OP) and 8-bit immediate data field (u8) or address designated field (rel8/dir8). In some instructions, it is 8-bit register list designated field (rlist).



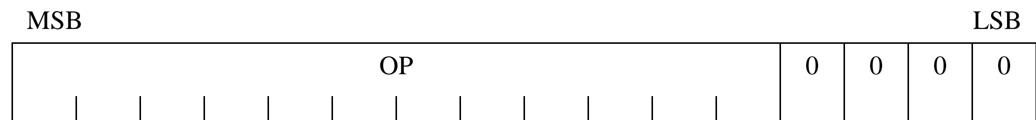
TYPE-E

It has 12-bit OP Codes (OP) and register designated fields (Ri/Rs).



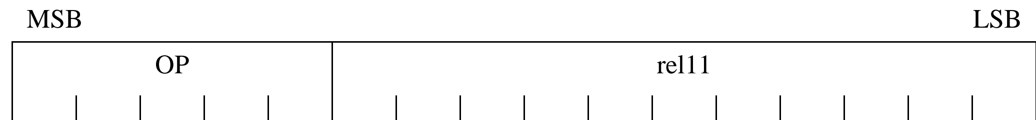
TYPE-E'

It is deformation of TYPE-E. It has 12-bit OP Code (OP). TYPE-E register designated field is fixed to 0000_B. It is applied for instructions where 16-bit instruction code such as NOP Instruction or RET Instructions etc. is defined.



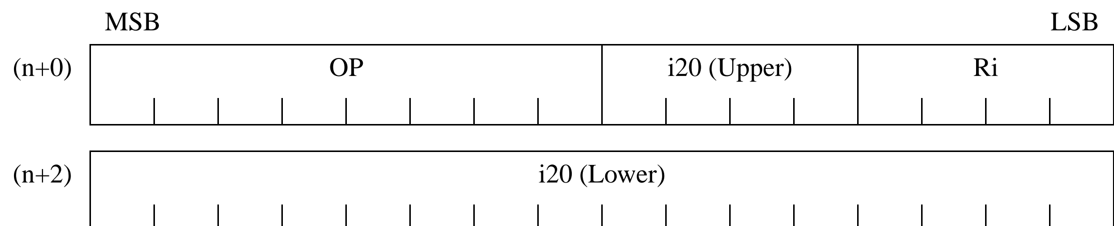
TYPE-F

It has 5-bit OP Code (OP) and 11-bit address designated filed (rel11).



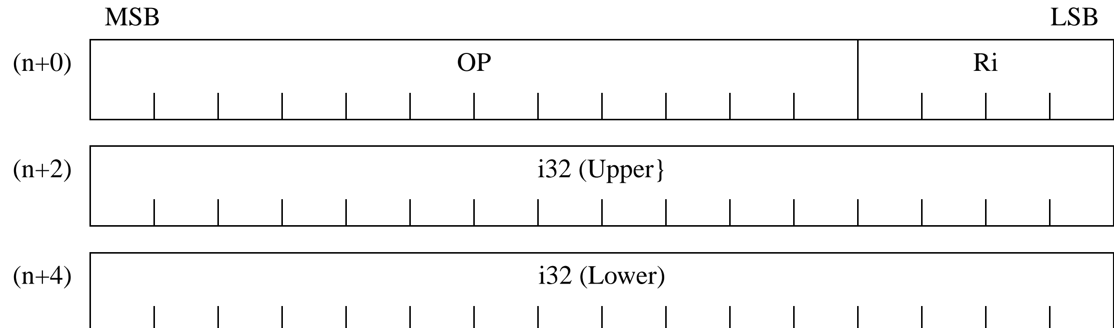
TYPE-G

It has 8-bit OP code (OP) and 20bit immediate data field (i20), register designated field (Ri). It has 32-bit length and is applied only for LDI:20 Instruction.



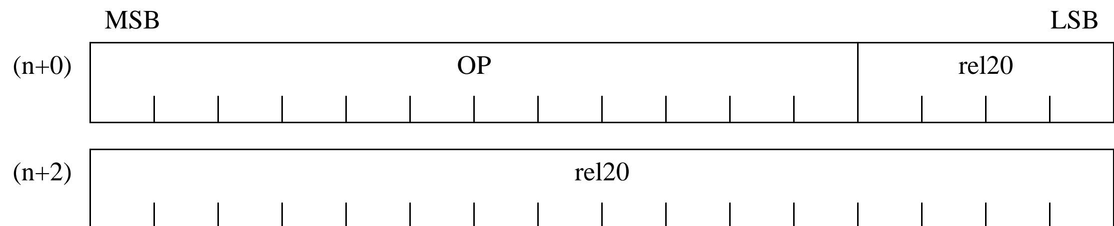
TYPE-H

It has 12bit OP code (OP) and 32bit immediate data field (i32), register designated field (Ri). It has 48-bit length and is applied only for LDI:32 Instruction.



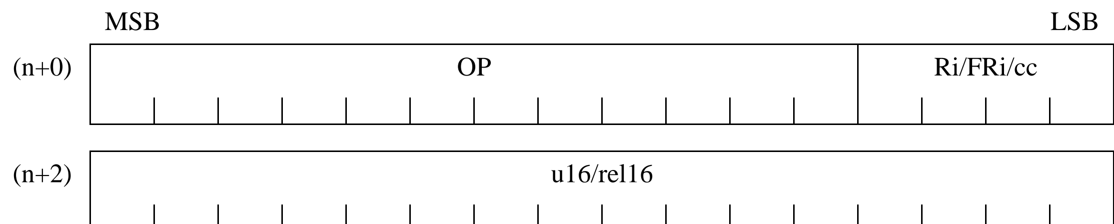
TYPE-I

It has 12-bit OP Code (OP) and 20-bit address designated filed (rel20). These are the instruction formats that have been added to FR81 Family CPU.



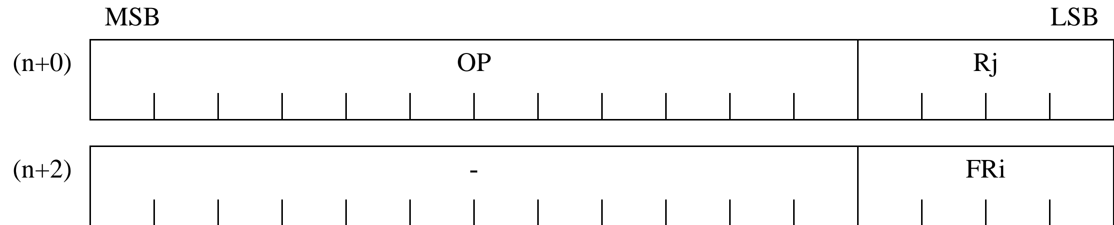
TYPE-J

It has 12-bit OP Code (OP), register designated fields (Rj/FRi/cc) and 16-bit address designated fields (u16/rel16). These are the instruction formats that have been added to FR81 Family CPU.



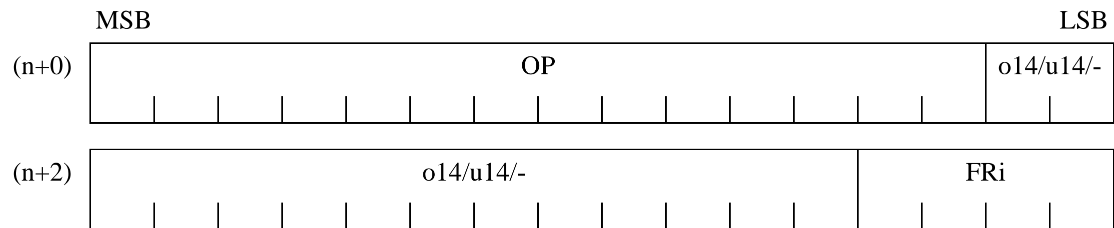
TYPE-K

It has 12-bit OP Code (OP), register designated field (Rj) and floating point register designated filed (FRi). These are the instruction formats that have been added to FR81 Family CPU.



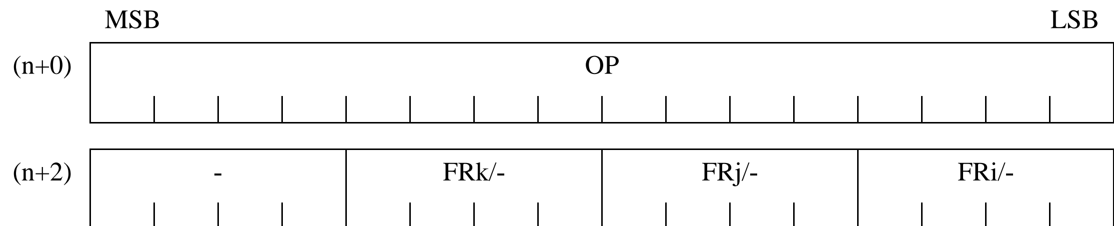
TYPE-L

It has 14-bit OP Code (OP) and 14-bit immediate data fields (o14/u14), floating point register designated field (FRi). Immediate data fields are not used in some instructions. These are the instruction formats that have been added to FR81 Family CPU.



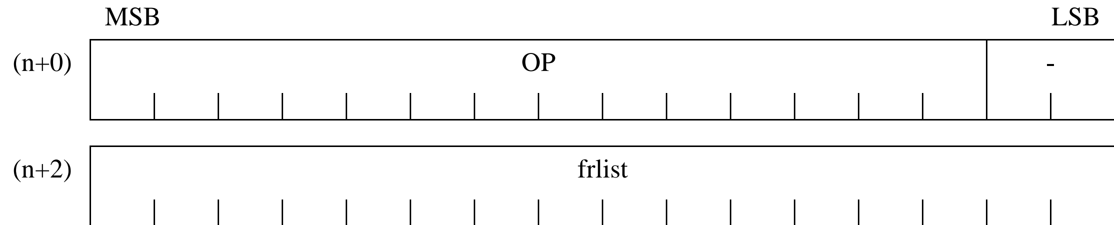
TYPE-M

It has 16-bit OP Code (OP) and three floating point register designated fields (FRk, FRj, FRi). These are the instruction formats that have been added to FR81 Family CPU.



TYPE-N

It has 14-bit OP Code (OP) and floating point register list (frlist). These are the instruction formats that have been added to FR81 Family CPU.



6.2.4 Register designated Field

● General-purpose register designated Field (Ri/Rj)

Among Instruction formats, fields that designate general-purpose register are 4-bit length Ri and Rj. Relation between bit pattern of general purpose register and register designated field has been indicated in Table 6.2-1.

Table 6.2-1 Bit pattern of general purpose register and register designated field

Ri / Rj	Register
0000_B	R0
0001_B	R1
0010_B	R2
0011_B	R3
0100_B	R4
0101_B	R5
0110_B	R6
0111_B	R7

Ri / Rj	Register
1000_B	R8
1001_B	R9
1010_B	R10
1011_B	R11
1100_B	R12
1101_B	R13
1110_B	R14
1111_B	R15

● Dedicated register designated Field (Rs)

Among Instruction formats, field that designates dedicated register is 4-bit length Rs. Relation between bit pattern of dedicated register and register designated field has been indicated in Table 6.2-2.

Table 6.2-2 Bit pattern of dedicated register and register designated field

Rs	Register	Rs	Register
0000_B	Table Base Register (TBR)	1000_B	Exception status register (ESR)
0001_B	Return Pointer (RP)	1001_B	Reserved
0010_B	System Stack Pointer (SSP)	1010_B	
0011_B	User Stack Pointer (USP)	1011_B	
0100_B	Multiply/Divide Register (MDH)	1100_B	
0101_B	Multiply/Divide Register (MDL)	1101_B	
0110_B	Base pointer (BP)	1110_B	
0111_B	FPU control register (FCR)	1111_B	Debug register (DBR)

Bit pattern which is shown as "Reserved" in the field that designates dedicated register is the reserved pattern. Operation when reserved pattern is specified is not covered by the warranty.

● Floating point register designated field

Among instruction formats, fields that designate floating point register are 4-bit length FRi, FRj and FRk. The relationship between the bit pattern of the floating point register and register designated field is indicated in Table 6.2-3.

Table 6.2-3 Bit pattern of floating point register and register designated field

FRk/FRj/FRi	Register	FRk/FRj/FRi	Register
0000_B	FR0	1000_B	FR8
0001_B	FR1	1001_B	FR9
0010_B	FR2	1010_B	FR10
0011_B	FR3	1011_B	FR11
0100_B	FR4	1100_B	FR12
0101_B	FR5	1101_B	FR13
0110_B	FR6	1110_B	FR14
0111_B	FR7	1111_B	FR15

FR81 Family

6.3 Data Format

This section describes the data type and format supported by FR81 Family CPU. In addition to integer type supported by FR80 and earlier, single precision floating point type has been added.

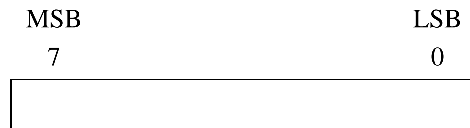
6.3.1 Data Format Used by Integer Type Instructions (Common with All FR Family)

● Signed integer byte

Signed integer byte is represented as consecutive 8 bits. Bit 7 represents the sign bit (S), and "0" represents positive or zero and "1" represents negative.

● Unsigned integer byte

Unsigned integer byte is represented as consecutive 8 bits.



● Signed integer half word

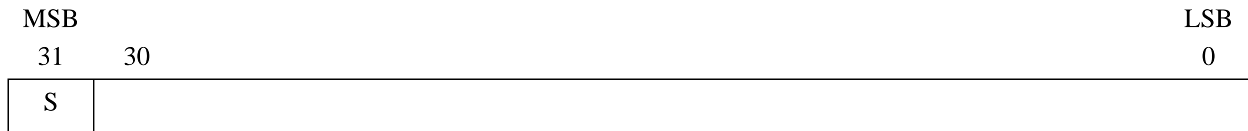
Signed integer half word is represented as consecutive 16 bits. Bit 15 represents the sign bit (S), and "0" represents positive or zero and "1" represents negative.

● Unsigned integer half word

Unsigned integer half word is represented as consecutive 16 bits.

● Signed integer word

Signed integer word is represented as consecutive 32 bits. Bit 31 represents the sign bit (S), and "0" represents positive or zero and "1" represents negative.



● Unsigned integer word

Unsigned integer word is represented as consecutive 32 bits.

6.3.2 Format Used for Floating Point Type Instructions

● Floating point format

The IEEE 754 standard is used for floating point format. A floating point is represented by the following 3 fields.

Field	Symbol	Content
Sign bit (Sign)	s	"0" represents positive, and "1" represents negative.
Exponent (Exponent)	e	Bias representation with single precision bias of 127, and the double precision bias of 1023
Fractional bit (Fraction)	f	The fraction represents a number less than 1, but the significant is 1 plus the fraction part.

Using the above-described symbols, floating point (normalized number) is represented by the following formula. Bias representation with single precision bias is 127, and the double precision bias is 1023.

$(-1)^{s} \times 1.f \times 2^{(e - bias)}$

In addition, there are special numbers of not-a-number (NaN), infinity (∞), zero and unnormalized number.

Signaling NaN (SNaN)	e - bias = Emax +1, and MSB of f is 0
Quiet NaN (QNaN)	e - bias = Emax +1, and MSB of f is 1
Infinity (+ ∞, - ∞)	e - bias = Emax +1, and f is 0
Normalized number	e - bias = Between Emin and Emax
Unnormalized number	e - bias = Emin -1, and f is not 0
Zero (+0, -0)	e, f = 0

- Single precision floating point (32-bit)

This conforms to IEEE754 single precision format and is represented as consecutive 32 bits. In the single precision floating point format, bit 31 represents the sign bit (S), bit 30 to bit 23 represent the exponent bits, and bit 22 to bit 0 represent the fractional bits.

6.4 Read-Modify-Write type Instructions

Read-Modify-Write type Instructions are those that carry out a series of operations namely, arithmetic processing in the data read from the memory space and write the result in the same address of the memory space.

IN registers of peripheral functions (I/O Registers), there are bits whose read values are different depending on instructions that independently carry out read access like LD Instruction and Read-Modify-Write type Instructions. Such bits have been described in the explanation on registers (I/O Registers) of peripheral functions.

In case of Read-Modify-Write type Instructions, a different instruction based on EIT processing is not executed between Read access and Write access of one instruction. This is used for exclusive control that uses flag or semaphore between programs.

Whether or not an instruction is Read-Modify-Write system Instruction is defined for each instruction. See "CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS".

FR81 Family

6.5 Branching Instructions and Delay Slot

FR81 Family CPU Branching Instructions are of two types namely, Delayed Branching Instructions and Non-delayed Branching Instructions.

6.5.1 Delayed Branching Instructions

In case of Delayed Branching Instructions, prior to execution of Branching Destination Instructions, instructions immediately after Branching Instructions are executed. Instructions immediately after Delayed Branching Instructions are called Delay slot.

Branching Instructions having ":D" affixed to mnemonic are Delayed Branching Instructions. Next Instruction will be Delayed Branching Instruction.

JMP:D	@Ri	CALL:D	label12	CALL:D	@Ri	RET:D	
BRA:D	label9	BNO:D	label9	BEQ:D	label9	BNE:D	label9
BC:D	label9	BNC:D	label9	BN:D	label9	BP:D	label9
BV:D	label9	BNV:D	label9	BLT:D	label9	BGE:D	label9
BLE:D	label9	BGT:D	label9	BLS:D	label9	BHI:D	label9

Since Delay Slot instructions are executed prior to Branching operation, apparent execution cycle of Branching Instruction will be 1 cycle. In case a valid instruction cannot be allocated in the Delay slot, it is necessary to allocate NOP Instruction. Example of Delayed Branching Instruction has been given below.

```
;       Instructions alignment
        ADD     R1,R2
        BRA:D   LABEL   ; Branching Instructions
        MOV     R2,R3   ; Delay slot (Executed before Branching)
        . . .
LABEL:  ST      R3,@R4  ; Branching Destination
```

In case of Conditional Branching Instruction, whether or not Branching conditions are established, Delay slot instructions are executed.

Instructions that can be placed in the Delay slot are only those that satisfy following conditions. When an attempt is made to execute an instruction that cannot be placed in the delay slot, an invalid instruction exception occurs and the EIT processing is carried out.

- 1 cycle Instructions
- Those that are not Branching Instructions

- Instructions that do not affect the operation even when the order is changed

1 cycle Instructions are those where anyone variable namely 1, a, b, c, d is independently written in the CYC Column of "A.2 Instruction Lists". (Instructions which have 2a or 1+b are not 1 cycle instructions).

List of instructions that can be placed in Delay Slot are indicated in "Appendix A.3".

EIT processing such as Step Trace Trap, general interrupt, NMI etc. cannot be accepted between Delayed Branching Instructions and Delay Slot Instructions.

6.5.2 Specific example of Delayed Branching Instructions

Specific example of Delayed Branching Instruction is given below.

"JMP:D @Ri" Instruction, "CALL:D @Ri" Instruction

General-purpose register Ri referred to during JMP:D Instruction, CALL:D Instruction is not affected by the Branching Destination Address even if Delay Slot Instruction updates Ri.

[Ex]

```
LDI:32  #Label,R0
JMP:D   @R0        ; Branching in Label
LDI:8   #0,R0      ; Does not affect Branching Destination Address
. . .
```

RET:D Instruction

Return Pointer (RP) referred to by RET:D Instruction is not affected even if Delay Slot Instruction updates the Return Pointer (RP).

[Ex]

```
RET:D              ; Branching to address indicated by RP set prior to this
MOV     R8,RP      ; Not affected by Return Operation
. . .
```

Bcc:D Instructions

Flag of Condition Code Register (CCR) referred to by Bcc:D Instruction is not affected by Delay Slot Instructions.

[Ex]

```
ADD      #1,R0      ; Flag change
BC:D     overflow   ; Branching based on execution result of preceding ADD Instruction
ANDCCR   #0         ; Updating of this flag does not affect Branching
```

CALL:D Instruction

If Return Pointer (RP) is referred to based on Delay Slot Instruction of CALL:D Instruction, updated content will be read based on CALL:D Instruction.

[Ex]

```
CALL:D   Label    ; Branching after updating of RP
MOV      RP,R0    ; Execution result of RP of preceding CALL:D Instruction is transferred to
                    R0
```

6.5.3 Non-Delayed Branching Instructions

In case of Non-Delayed Branching Instructions, execution is carried out in the sequence of Instructions. Instruction immediately after Branching Instruction is never executed before branching.

Branching Instructions without ":D" in mnemonic are Non-Delayed Branching Instructions. Next instruction will be Non-Delayed Branching Instruction

```
JMP   @Ri       CALL  label12    CALL  @Ri       RET
BRA   label9    BNO   label9     BEQ   label9    BNE  label9
BC    label9    BNC   label9     BN    label9    BP   label9
BV    label9    BNV   label9     BLT   label9    BGE  label9
BLE   label9    BGT   label9     BLS   label9    BHI  label9
```

Execution cycles of Non-Delayed Branching Instruction will be 2 cycles when Branching and 1 cycle when not branching. Example of Non-Delayed Branching Instruction is given below.

```
;         Sequence of Instructions
          ADD     R1,R2
          BRA     LABEL     ; Branching Instruction
          MOV     R2,R3     ; Not executed
          . . .
LABEL:    ST      R3,@R4    ; Branching Destination
```

Compared to Delayed Branching Instructions where NOP Instruction is placed in the Delay Slot, efficiency of instruction code can be increased. Execution speed and Code Efficiency both can be realized by using Delayed Branching Instruction when valid instruction can be placed in the Delay Slot and using Non-delayed Branching Instruction otherwise.

6.6 Step Division Instructions

In FR81 Family CPU, 32-bit signed/unsigned division is carried out based on combination of Step Division Instructions.

Step Division Instructions are of following types.

- DIV0S (Initial Setting Up for Signed Division)
- DIV0U (Initial Setting Up for Unsigned Division)
- DIV1 (Main Process of Division)
- DIV2 (Correction When Remain is zero)
- DIV3 (Correction When Remain is zero)
- DIV4S (Correction Answer for Signed Division)

In order to realize signed division, combine the Instructions as follows.

DIV0S, DIV1 × 32, DIV2, DIV3, DIV4S

In order to realize unsigned division, combine the Instructions as follows.

DIV0U, DIV1 × 32

For various Instructions, see "CHAPTER 7 DETAILED EXECUTION INSTRUCTIONS".

6.6.1 Signed Division

Signed 32bit dividend is divided with signed 32 bit divisor and quotient of signed 32 bit and remainder of signed 32bit are obtained.

Before carrying out division, dividend and divisor are set in the following register.

- Multiplication/Division Register (MDL): Dividend of signed 32 bit (Dividend)
- One of general-purpose registers: Divisor of signed 32 bit (Divisor)

Signed division is carried out by executing following 36 Instructions. DIV1 Instructions 32 numbers are arranged after DIV0S Instructions. In the operand of DIV0S Instructions, DIV1 Instructions, DIV2 Instructions general-purpose registers that store divisor are specified.

```
DIV0S   R2    ; Divisor in R2
DIV1    R2    ; #1
DIV1    R2    ; #2
. . .
DIV1    R2    ; #30
```

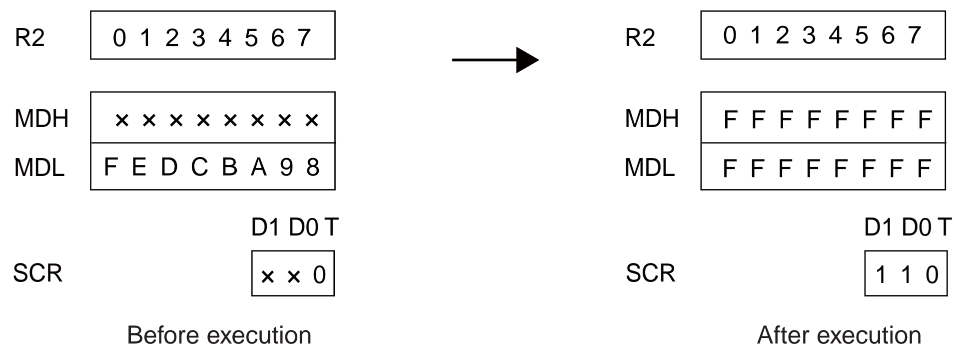
```
DIV1    R2    ; #31
DIV1    R2    ; #32
DIV2    R2
DIV3
DIV4S
```

Division results are stored in the following registers.

- Multiplication/Division Register (MDL): quotient of signed 32 bit
- Multiplication/Division Register (MDH): remainder of signed 32 bit

Example of execution of signed division has been indicated in Figure 6.6-1.

Figure 6.6-1 Example of execution of signed division



In SOFTUNE Assembler, DIV Instruction has been arranged to carry out signed division as Assembler Pseudo Machine Instruction. Using this DIV Instruction in place of above mentioned 36 Instructions, signed division can be described with 1 Instruction. See "FR family SOFTUNE Assembler Manual" for DIV Instruction.

6.6.2 Unsigned Division

Dividend of unsigned 32 bit dividend is divided with unsigned 32bit divisor and quotient of unsigned 32 bit and remainder of unsigned 32 bit are obtained.

Before carrying out division, dividend and divisor are set in the following register.

- Multiplication/Division Register (MDL): Dividend of unsigned 32 bit (Dividend)
- One of general-purpose registers: Divisor of unsigned 32 bit (Divisor)

Unsigned division is carried out by executing following 33 Instructions. DIV1 Instructions 32 numbers are arranged after DIV0U Instruction. In the operand of DIV0U Instructions, DIV1 Instructions, general-purpose registers that store divisor are specified.

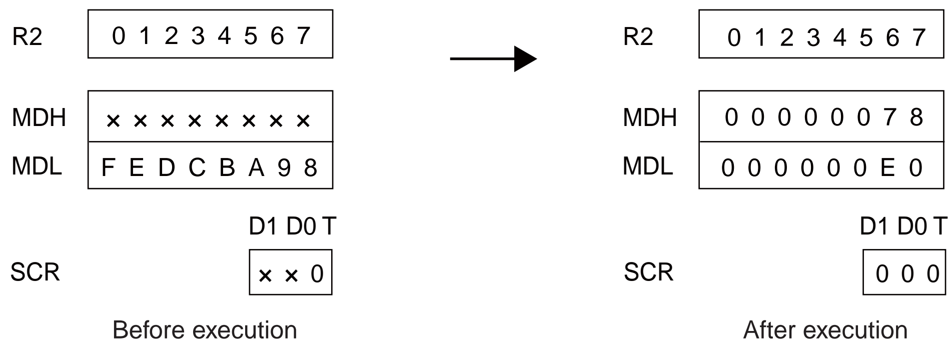
```
DIV0S   R2   ; divisor in R2
DIV1    R2   ; #1
DIV1    R2   ; #2
. . .
DIV1    R2   ; #30
DIV1    R2   ; #31
DIV1    R2   ; #32
```

Division result is stored in the following registers.

- Multiplication/Division Register (MDL): quotient of unsigned 32 bit
- Multiplication/Division Register (MDH): remainder of unsigned 32 bit

Example of execution of unsigned division has been indicated in Figure 6.6-2.

Figure 6.6-2 Example of execution of unsigned division



In SOFTUNE Assembler, DIVU Instruction has been arranged to carry out unsigned division as Assembler Pseudo Machine Instruction. Using this DIVU Instruction in place of above mentioned 33 Instructions, unsigned division can be described with 1 Instruction. See "FR family SOFTUNE Assembler Manual" for DIVU Instruction.

CHAPTER 7

DETAILED EXECUTION INSTRUCTIONS

This chapter explains each of the execution instructions used by the FR81 Family CPU, alphabetically in the reference format.

Refer to "A.1 Meaning of Symbols" for explanation regarding symbols used in detailed execution instructions. The respective instructions are explained separately in the following items.

- Assembler Format

Shows the format of writing the instruction in the assembler language.

- Operation

Shows the operation of an instruction by substituting it with an arrow mark (→).

- Flag Change

Shows whether the flag of the Condition Code Register (CCR) changes by the execution of an instruction.

- Classification

Shows Functional classification of instructions and the following sections of instructions.

Instruction with delay slot: Instruction than can be positioned in the delay slot

Read-Modify-Write system Instruction

FR80 Family: Instructions added to FR80 Family and after CPUs

FR81 Family: Instructions added in FR81 Family CPU

FR81 Updating: Instruction to which definition is changed in FR81 family CPU

- Execution Cycle

Shows the required number of clock cycles for instruction execution

- Instruction Format

Shows the format and bit pattern of the instruction.

- Execution example

Shows the operation example at the time of instruction execution.

7.1 ADD (Add 4bit Immediate Data to Destination Register)

Adds the result of higher 28 bits of 4-bit immediate data with zero extension(0-15) and stores the results to Ri.

- Assembler Format

ADD #i4, Ri

- Operation

Ri + extu(i4) → Ri

- Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a carry has occurred as a result of the operation, cleared otherwise.

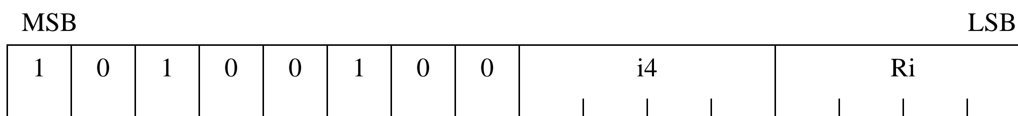
- Classification

Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles

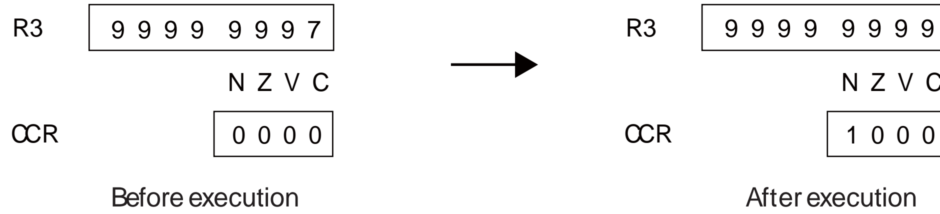
1 cycle

- Instruction Format



● Execution Example

ADD #2, R3 ; Bit pattern of instruction: 1010 0100 0010 0011



FR81 Family

7.2 ADD (Add Word Data of Source Register to Destination Register)

Adds word data of Rj to Ri, stores result to Ri.

- Assembler Format

ADD   Rj, Ri

- Operation

Ri + Rj → Ri

- Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a carry has occurred as a result of the operation, cleared otherwise.

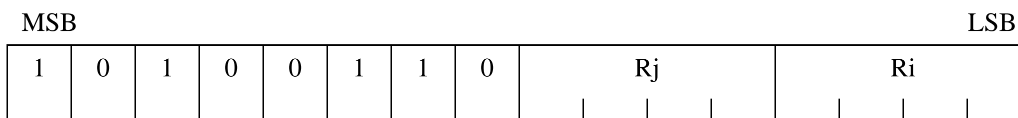
- Classification

Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles

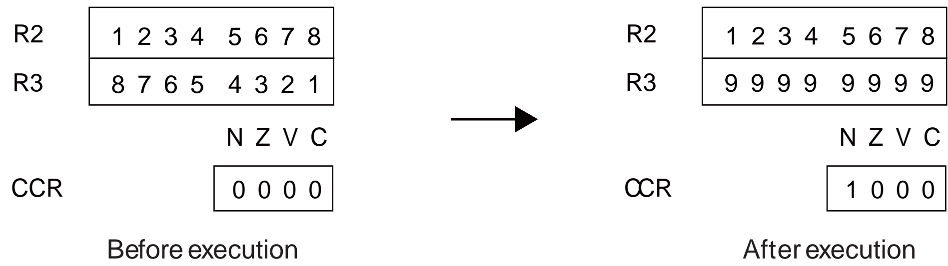
1 cycle

- Instruction Format



● Execution Example

ADD R2, R3 ; Bit pattern of instruction: 1010 0110 0010 0011



7.3 ADD2 (Add 4bit Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with minus extension (-16 to -1) to word data in Ri, stores results to Ri. Unlike SUB instruction, changing C flag of this instruction becomes it as well as the ADD instruction unlike the SUB instruction.

- Assembler Format

ADD2 #i4, Ri

- Operation

Ri + extn(i4) → Ri

- Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a carry has occurred as a result of the operation, cleared otherwise.

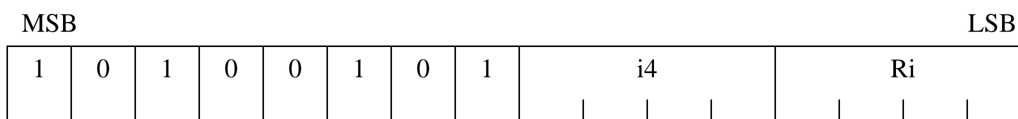
- Classification

Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles

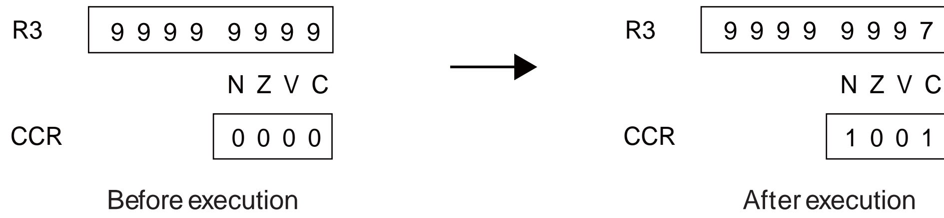
1 cycle

- Instruction Format



● Execution Example

ADD2 #-2, R3 ; Bit pattern of instruction: 1010 0101 1110 0011



7.4 ADDC (Add Word Data of Source Register and Carry Bit to Destination Register)

Adds word data and carry flag (C) of Rj to Ri, stores results in Ri.

- Assembler Format

ADDC Rj, Ri

- Operation

Ri + Rj + C → Ri

- Flag Change

N	Z	V	C
C	C	C	C

N: Set when MSB of the operation result is "1", cleared when MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a carry has occurred as a result of the operation, cleared otherwise.

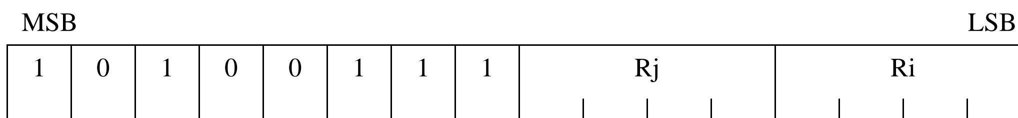
- Classification

Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles

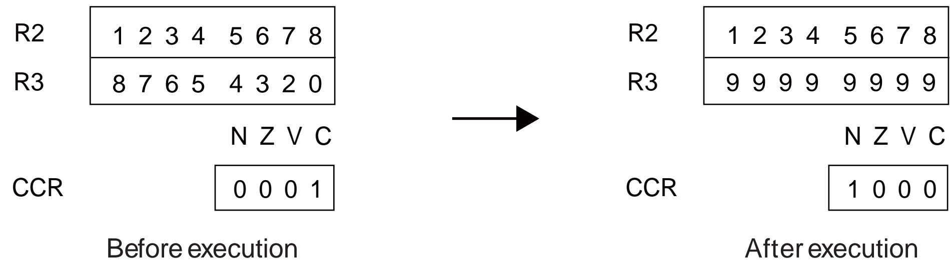
1 cycle

- Instruction Format



● Execution Example

ADDC R2, R3 ; Bit pattern of the instruction: 1010 0111 0010 0011



FR81 Family

7.5 ADDN (Add Immediate Data to Destination Register)

Adds the result of the higher 28 bits of the 4-bit immediate data with zero extension (0 to 15) to the word data of Ri, stores the results without changing flag settings.

- Assembler Format

ADDN #i4, Ri

- Operation

Ri + extu(i4) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

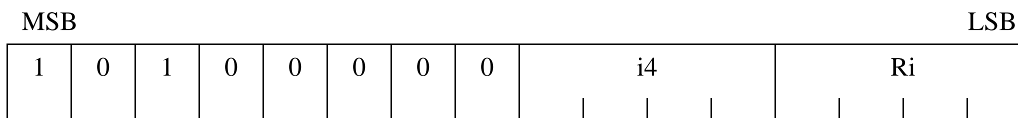
- Classification

Add/Subtract Instruction, Instruction with delay slot

- Execution Cycles

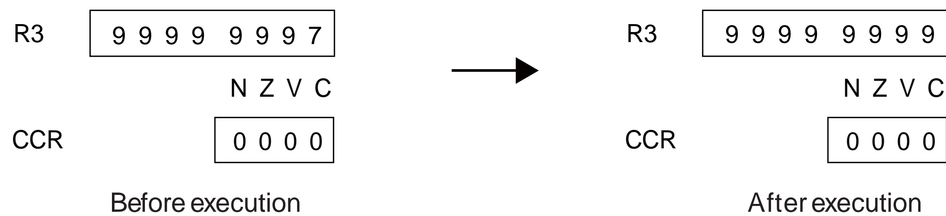
1 cycle

- Instruction Format



- Execution Example

ADDN #2, R3 ; Bit pattern of the instruction: 1010 0000 0010 0011



7.6 ADDN (Add Word Data of Source Register to Destination Register)

Adds the word data of Rj to the word data of Ri, stores results in Ri without changing flag settings.

● Assembler Format

ADDN Rj, Ri

● Operation

Ri + Rj → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

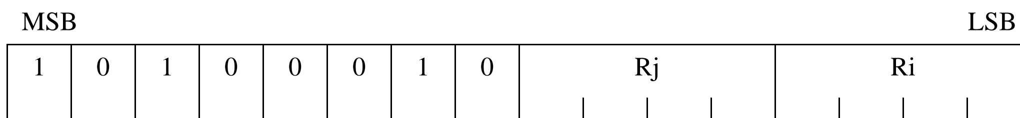
● Classification

Add/Subtract Instruction, Instruction with delay slot

● Execution Cycles

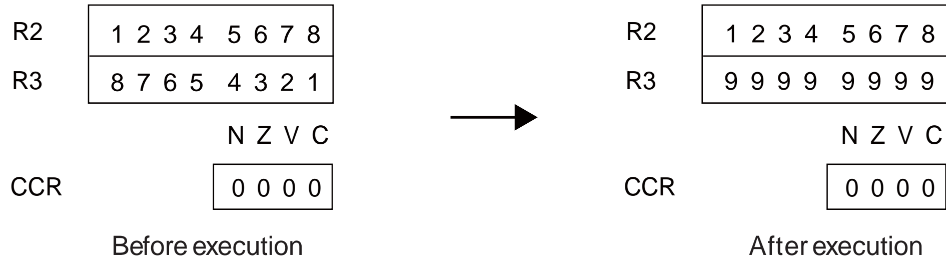
1 cycle

● Instruction Format



● Execution Example

ADDN R2, R3 ; Bit pattern of the instruction: 1010 0010 0010 0011



FR81 Family

7.7 ADDN2 (Add Immediate Data to Destination Register)

Adds the result of the higher 28 bits of 4-bit immediate data with minus extension (-16 to -1) to word data in Ri, stores the results in Ri without changing flag settings.

● Assembler Format

ADDN2 #i4, Ri

● Operation

Ri + extn(i4) → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

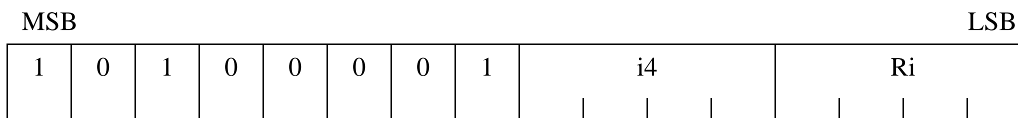
● Classification

Add/Subtract Instruction, Instruction with delay slot

● Execution Cycles

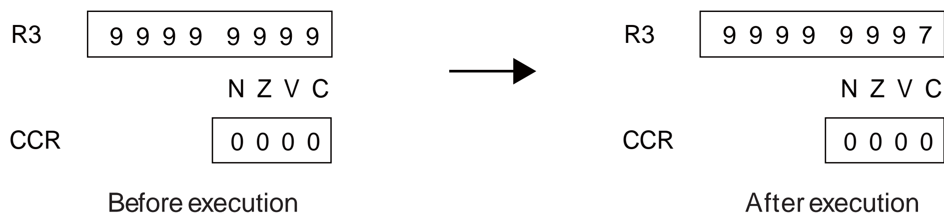
1 cycle

● Instruction Format



● Execution Example

ADDN2 #-2, R3 ; Bit pattern of the instruction: 1010 0001 1110 0011



7.8 ADDSP (Add Stack Pointer and Immediate Data)

Adds 4 times the 8-bit immediate data as a signed extended value to the word data of R15 and stores result in R15. Specifies the value of s8 × 14 as s10.

- Assembler Format

ADDSP #s10

- Operation

R15 + exts(s8×4) → R15

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

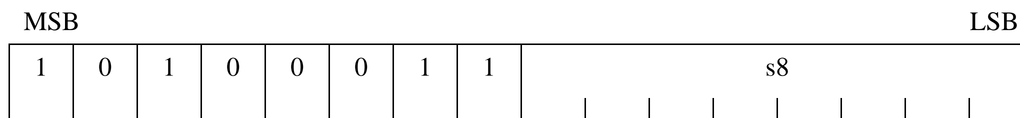
- Classification

Other instructions, Instruction with delay slot

- Execution Cycles

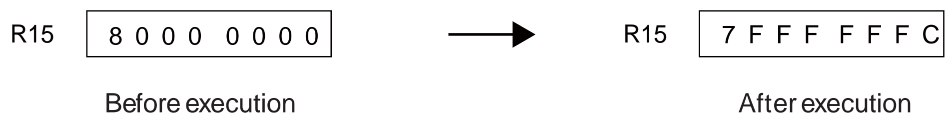
1 cycle

- Instruction Format



● Execution Example

ADDSP #-4 ; Bit pattern of the instruction: 1010 0011 1111 1111



7.9 AND (And Word Data of Source Register to Data in Memory)

Takes the logical AND of the word data at memory address Ri and word data in Rj and stores the results to the memory address corresponding to Ri.

- Assembler Format

AND Rj,@Ri

- Operation

(Ri) & Rj → (Ri)

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

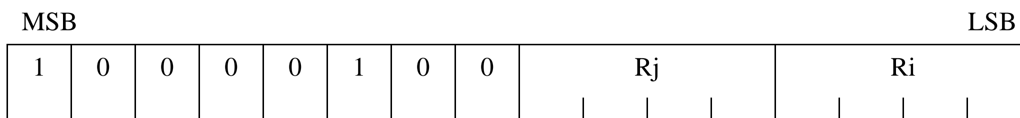
- Classification

Logical calculation instruction, Read/Modify/Write type instruction

- Execution Cycle

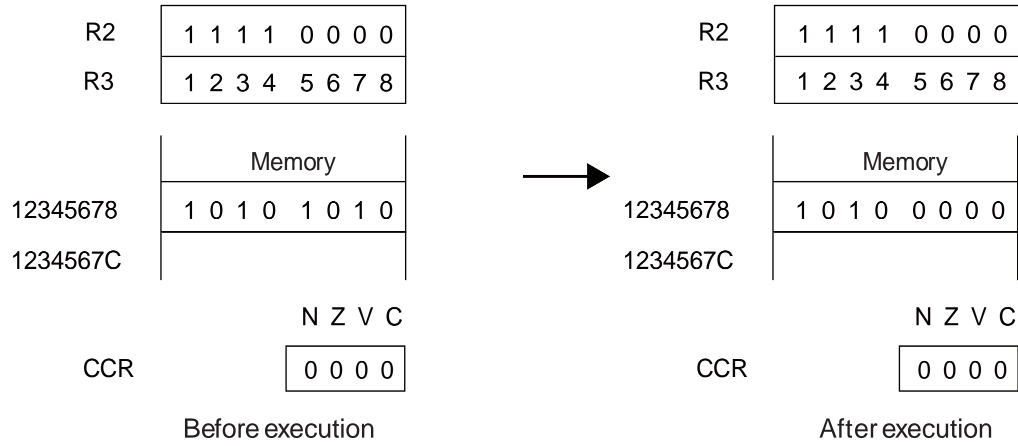
1+2a cycles

- Instruction Format



- Execution Example

```
AND   R2,@R3       ; Bit pattern of the instruction: 1000 0100 0010 0011
```



7.10 AND (And Word Data of Source Register to Destination Register)

Takes the logical AND of word data in Ri and word data in Rj and stores the results to Rj.

- Assembler Format

AND Rj, Ri

- Operation

Ri & Rj → Ri

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "“0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

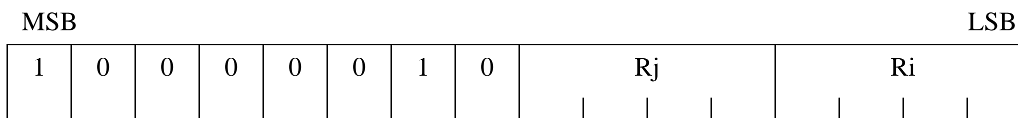
- Classification

Logical calculation instruction, Instruction with delay slot

- Execution Cycles

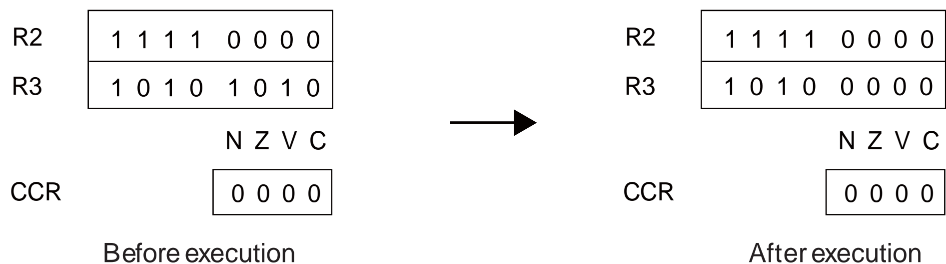
1 cycle

- Instruction Format



● Execution Example

AND R2, R3 ; Bit pattern of the instruction: 1000 0010 0010 0011



FR81 Family

7.11 ANDB (And Byte Data of Source Register to Data in Memory)

Takes the logical AND of the byte data at memory address Ri and the byte data in Rj and stores the results at Ri location in the memory.

- Assembler Format

ANDB Rj,@Ri

- Operation

(Ri) & Rj → (Ri)

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

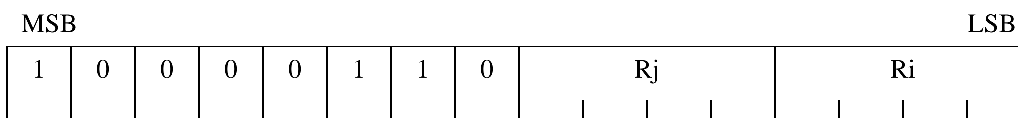
- Classification

Logical calculation instruction, Read/Modify/Write type instruction

- Execution Cycles

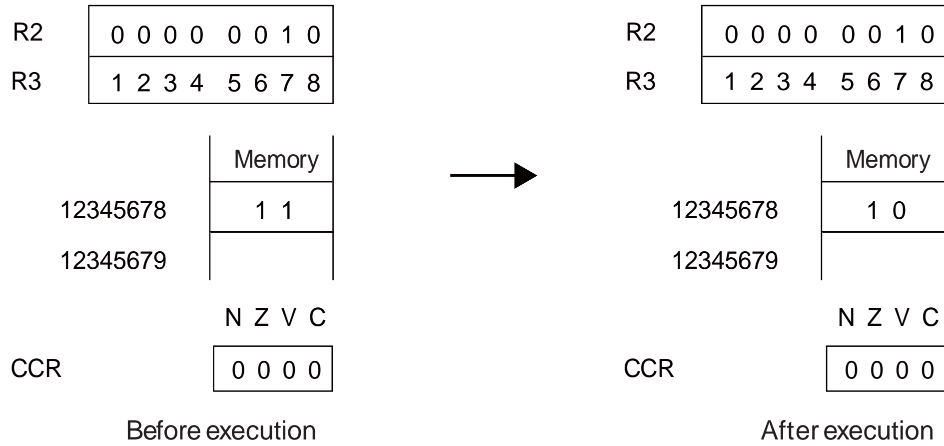
1+2a cycles

- Instruction Format



● Execution Example

ANDB R2,@R3 ; Bit pattern of the instruction: 1000 0110 0010 0011



7.12 ANDCCR (And Condition Code Register and Immediate Data)

Takes the logical AND of byte data in the condition code register (CCR) and 8-bit immediate data and returns the results to the CCR.

● Assembler Format

ANDCCR #u8

● Operation

User mode:

CCR & (u8 | 30H) → CCR

Privilege mode

CCR & u8 → CCR

In user mode, a request to rewrite the stack flag (S) or the interrupt enable flag (I) is ignored. The S and I flags can only be changed in privilege mode.

● Flag Change

S	I	N	Z	V	C
C	C	C	C	C	C

S, I, N, Z, V, C: Varies according to results of operation.

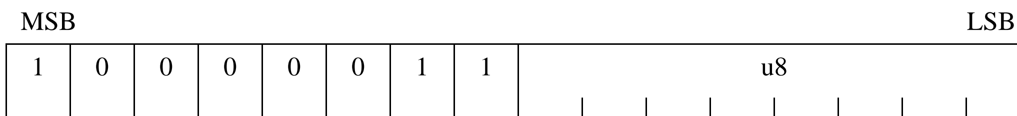
● Classification

Other instructions, Instruction with delay slot, FR81 updating

● Execution Cycles

1 cycle

● Instruction Format

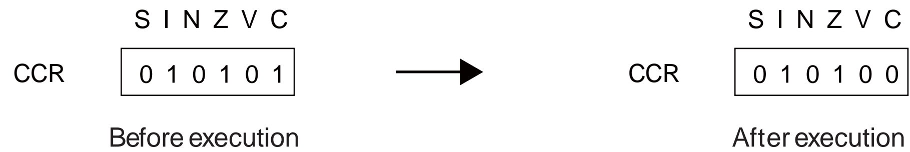


● EIT Occurrence and Detection

An interrupt is detected (the value of I flag after instruction execution is used).

● Execution Example

ANDCCR #0FEH ; Bit pattern of the instruction: 1000 0011 1111 1110



7.13 ANDH (And Halfword Data of Source Register to Data in Memory)

Takes the logical AND of the half-word data at Ri location of the memory and the half-word data in Rj and stores the results at Ri location of the memory.

- Assembler Format

ANDH   Rj,@Ri

- Operation

(Ri) & Rj → (Ri)

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB (bit15) is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

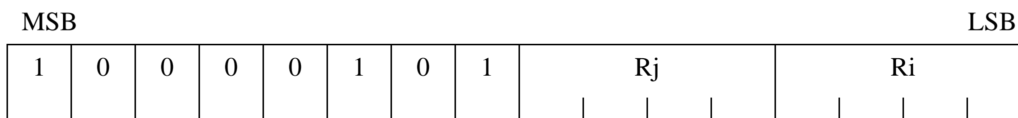
- Classification

Logical calculation instruction, Read/Modify/Write type instruction

- Execution Cycles

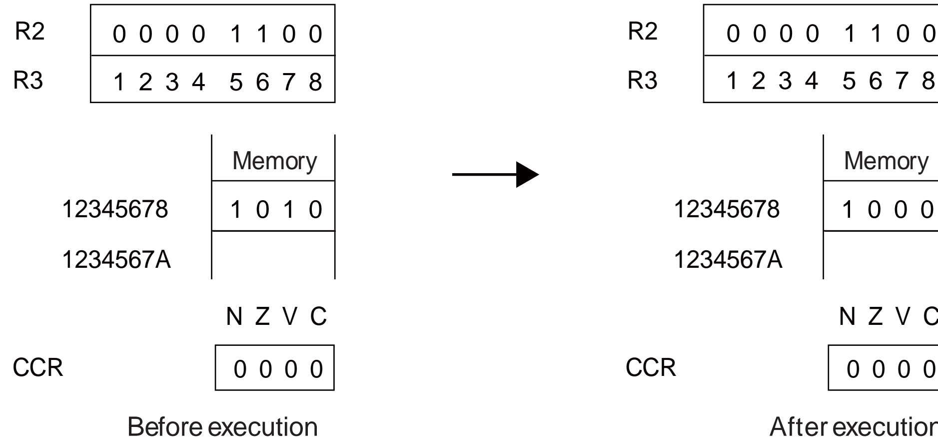
1+2a cycles

- Instruction Format



- Execution Example

ANDH R2,@R3 ; Bit pattern of the instruction: 1000 0101 0010 0011



7.14 ASR (Arithmetic shift to the Right Direction)

Makes an arithmetic right shift of the word data in Ri by Rj bits, stores the result to Ri. Only the lower 5 bits of Rj, which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

● Assembler Format

ASR Rj, Ri

● Operation

Ri >> Rj → Ri

● Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last. Cleared when the shift amount is "0".

● Classification

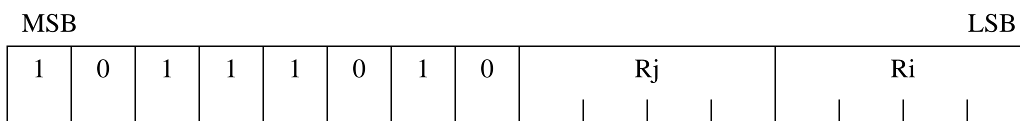
Shift instructions, Instruction with delayed slot

● Execution Cycles

1 cycle

● Instruction Format

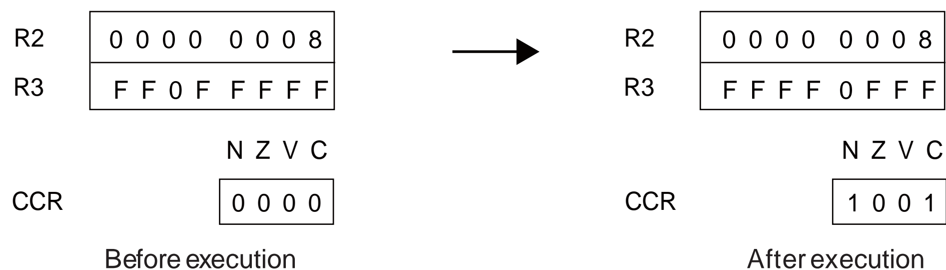
MSB LSB



1	0	1	1	1	0	1	0	Rj	Ri

● Execution Example

ASR R2, R3 ; Bit pattern of the instruction: 1011 1010 0010 0011



7.15 ASR (Arithmetic shift to the Right Direction)

Makes an arithmetic right shift of the word data in Ri by u4 bits, stores the result to Ri.

- Assembler Format

ASR #u4, Ri

- Operation

Ri >> u4 → Ri

- Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last. Cleared when the shift amount is "0".

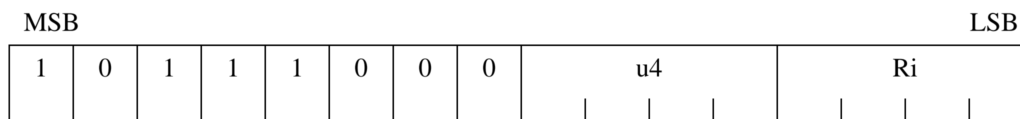
- Classification

Shift instruction, Instruction with delay slot

- Execution Cycles

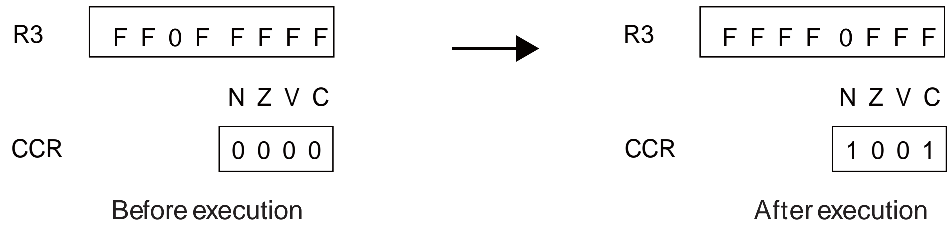
1 cycle

- Instruction Format



- Execution Example

ASR #8, R3 ; Bit pattern of the instruction: 1011 1000 1000 0011



7.16 ASR2 (Arithmetic shift to the Right Direction)

Makes an arithmetic right shift of the word data in Ri by u4+16 bits, stores the result to Ri.

● Assembler Format

ASR2 #u4, Ri

● Operation

Ri >> {u4 + 16} → Ri

● Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last.

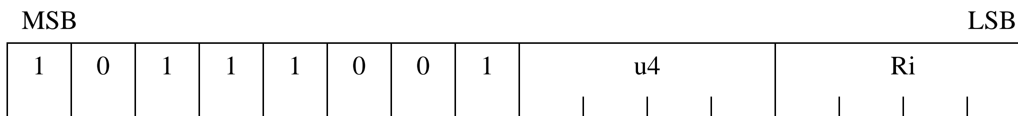
● Classification

Shift instruction, Instruction with delayed slot

● Execution Cycles

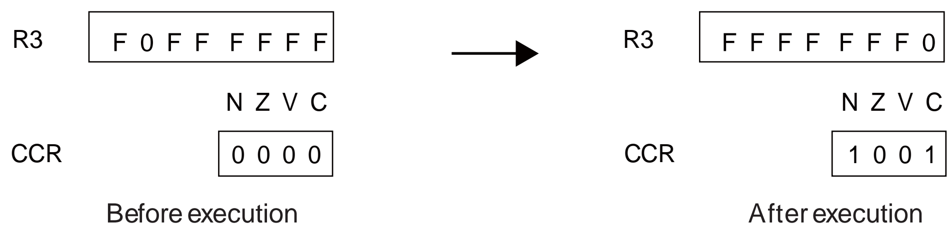
1 cycle

● Instruction Format



● Execution Example

ASR2 #8, R3 ; Bit pattern of the instruction: 1011 1001 1000 0011



7.17 BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the higher 4 bits of byte data at memory Ri, stores the results to the memory address corresponding to Ri.

- Assembler Format

BANDH #u4,@Ri

- Operation

(Ri) & {u4 << 4 + $0F_H$} → (Ri) [Operation uses higher 4 bits only]

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

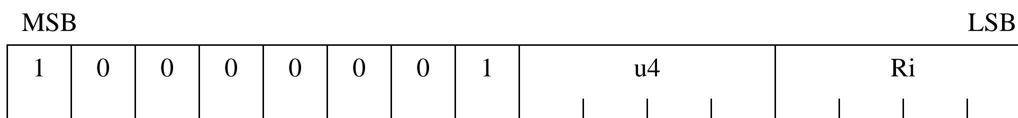
- Classification

Bit operation instruction, Read/Modify/Write type instruction

- Execution Cycles

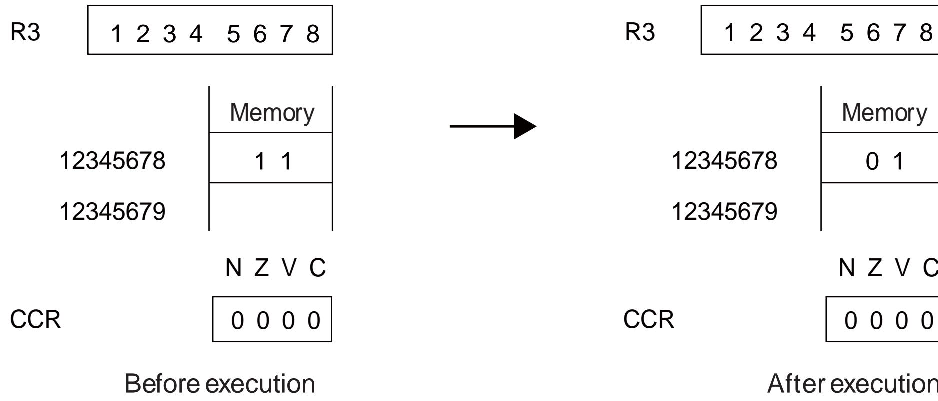
1+2a cycles

- Instruction Format



- Execution Example

BANDH #0,@R3 ; Bit pattern of the instruction: 1000 0001 0000 0011



7.18 BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the lower 4 bits of byte data at memory Ri, stores the results to the memory address corresponding to Ri.

● Assembler Format

BANDL #u4,@Ri

● Operation

(Ri) & {$F0_H$ + u4} → (Ri) [Operation uses lower 4 bits only]

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

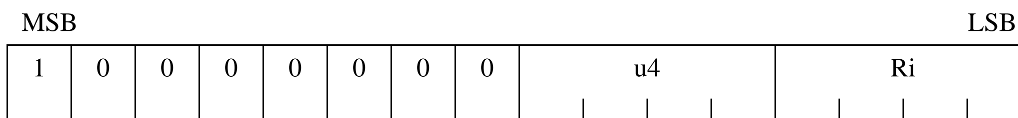
● Classification

Bit operation instructions, Read/Modify/Write type instruction

● Execution Cycles

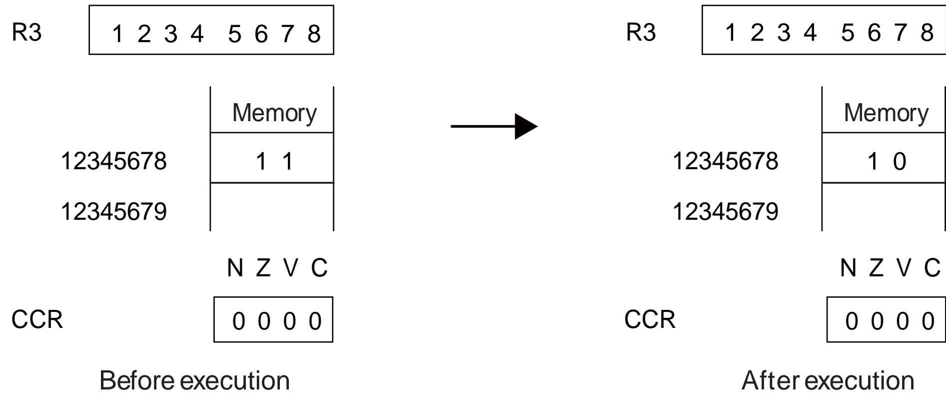
1+2a cycles

● Instruction Format



● Execution Example

BANDL #0,@R3 ; Bit pattern of the instruction: 1000 0000 0000 0011



FR81 Family

7.19 Bcc (Branch relative if Condition satisfied)

This is a branching instruction without a delay slot. If the conditions specified for each instruction are satisfied, branch to the address indicated by label9 relative to the value of the program counter (PC). When calculating the address, double the value of rel8 as a signed extension. If conditions are not satisfied, no branching occurs.

- Assembler Format

BRA	label9	BV	label9
BNO	label9	BNV	label9
BEQ	label9	BLT	label9
BNE	label9	BGE	label9
BC	label9	BLE	label9
BNC	label9	BGT	label9
BN	label9	BLS	label9
BP	label9	BHI	label9

- Operation

if (condition) then

$PC + 2 + exts(rel8 \times 2) \rightarrow PC$

Branching of each instruction is shown in Table 7.19-1.

Table 7.19-1 Branching conditions

Mnemonic	cc	Condition	Mnemonic	cc	Condition
BRA	0000	Always satisfied	BV	1000	V == 1
BNO	0001	Always unsatisfied	BNV	1001	V == 0
BEQ	0010	Z == 1	BLT	1010	(V ^ N) == 1
BNE	0011	Z == 0	BGE	1011	(V ^ N) == 0
BC	0100	C == 1	BLE	1100	((V ^ N) | Z) == 1
BNC	0101	C == 0	BGT	1101	((V ^ N) | Z) == 0
BN	0110	N == 1	BLS	1110	(C | Z) == 1
BP	0111	N == 0	BHI	1111	(C | Z) == 0

| : Logical add (or) ^ : Exclusive-OR (exor) ==: comparison operation (satisfied by congruence)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

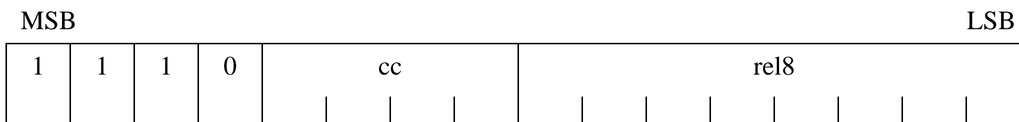
Non-delayed branching instruction

● Execution Cycles

At time of branching: 2 cycles

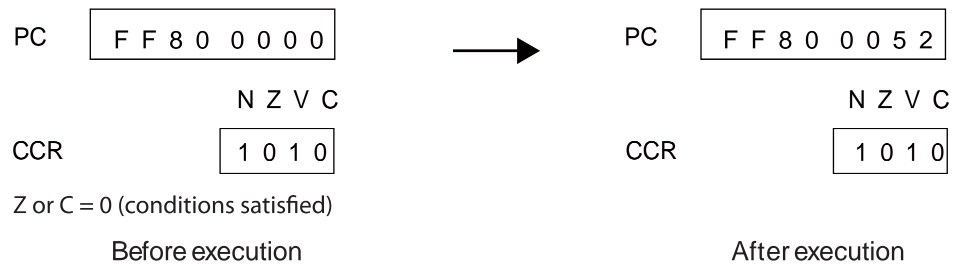
At the time of no branching: 1 cycle

● Instruction Format



● Execution Example

```
BHI   label        ; Bit pattern of the instruction: 1110 1111 0010 1000
. . .
label:             ; Address of BHI Instruction + 50H
```



FR81 Family

7.20 Bcc:D (Branch relative if Condition satisfied)

This is a branching instruction with a delay slot. If the conditions established for each particular instruction are satisfied, branch to the address indicated by label9 relative to the value of the program counter (PC). When calculating the address, double the value of rel8 as a signed extension. If conditions are not satisfied, no branching occurs.

● Assembler Format

```
BRA:D  label9        BV:D   label9
BNO:D  label9        BNV:D  label9
BEQ:D  label9        BLT:D  label9
BNE:D  label9        BGE:D  label9
BC:D   label9        BLE:D  label9
BNC:D  label9        BGT:D  label9
BN:D   label9        BLS:D  label9
BP:D   label9        BHI:D  label9
```

● Operation

if (condition) then

PC + 2 + exts(rel8 × 2) → PC

Branching conditions of each instruction are shown in Table 7.20-1.

Table 7.20-1 Branching conditions

Mnemonic	cc	Condition	Mnemonic	cc	Condition
BRA:D	0000	Always satisfied	BV:D	1000	V == 1
BNO:D	0001	Always unsatisfied	BNV:D	1001	V == 0
BEQ:D	0010	Z == 1	BLT:D	1010	(V ^ N) == 1
BNE:D	0011	Z == 0	BGE:D	1011	(V ^ N) == 0
BC:D	0100	C == 1	BLE:D	1100	((V ^ N)	Z) == 1
BNC:D	0101	C == 0	BGT:D	1101	((V ^ N)	Z) == 0
BN:D	0110	N == 1	BLS:D	1110	(C | Z) == 1
BP:D	0111	N == 0	BHI:D	1111	(C | Z) == 0

| : Logical add (or) ^ : Exclusive-OR (exor)   ==: comparison operation (satisfied by congruence)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

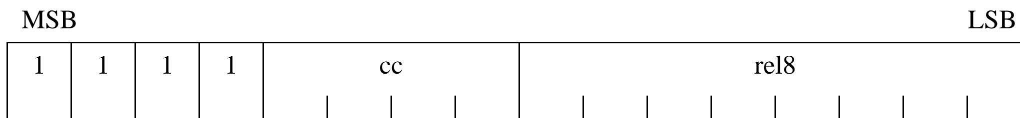
● Classification

Delayed branching instruction

● Execution Cycles

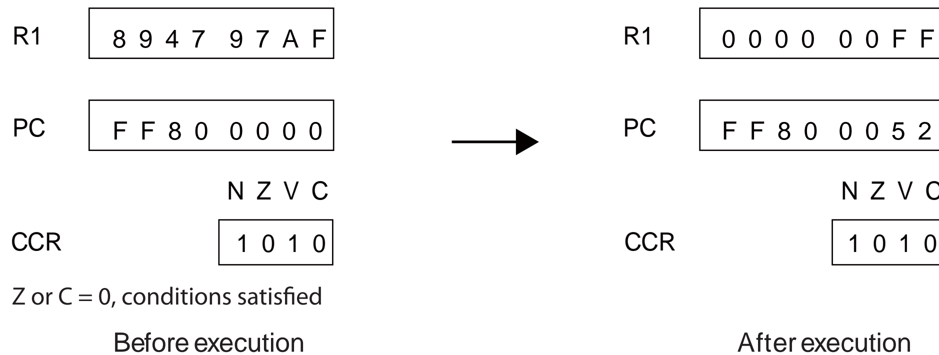
1 cycle

● Instruction Format



● Execution Example

```
BHI:D   label      ; Bit pattern of the instruction: 1111 1111 0010 1000
LDI:8   #255, R1   ; Instruction placed in delay slot
. . .
label:             ; BHI:D instruction address + 50H
```



The instruction placed in delay slot will be executed before the execution of the branch destination instruction. The value of R1 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.

7.21 BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory)

Takes the logical exclusive OR of the 4-bit immediate data and the higher 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

- Assembler Format

BEORH #u4,@Ri

- Operation

(Ri) ^ {u4 << 4} → (Ri) [Operation uses higher 4 bits only]

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

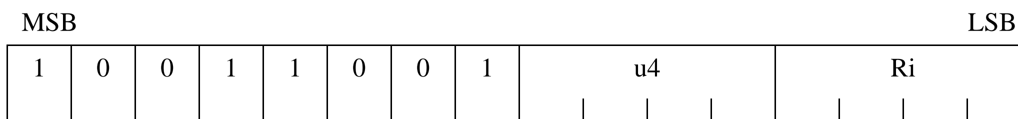
- Classification

Bit Operation instruction, Read/Modify/Write type instruction

- Execution Cycles

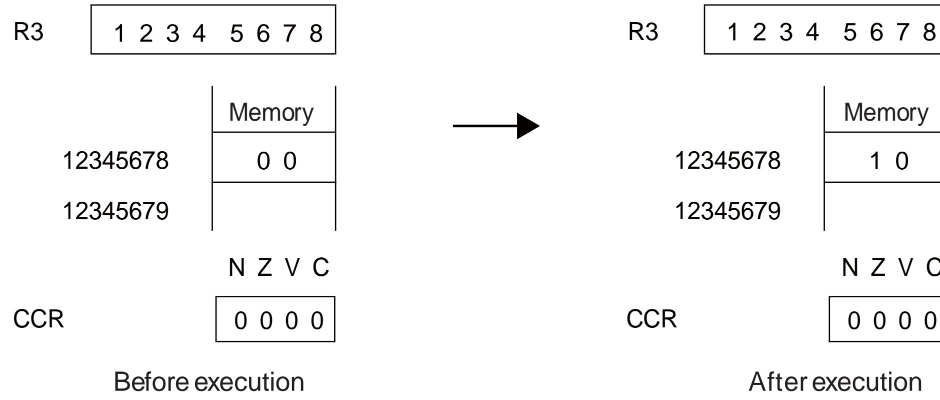
1+2a cycles

- Instruction Format



● Execution Example

BEORH #1,@R3 ; Bit pattern of the instruction: 1001 1001 0001 0011



FR81 Family

7.22 BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory)

Takes the logical exclusive OR of the 4-bit immediate data and the lower 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

- Assembler Format

BEORL #u4,@Ri

- Operation

(Ri) ^ u4 → (Ri) [Operation uses lower 4 bits only]

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

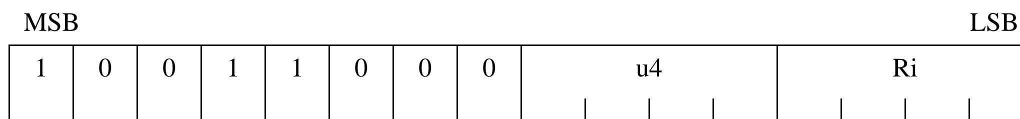
- Classification

Bit Operation instruction, Read/Modify/Write type instruction

- Execution Cycles

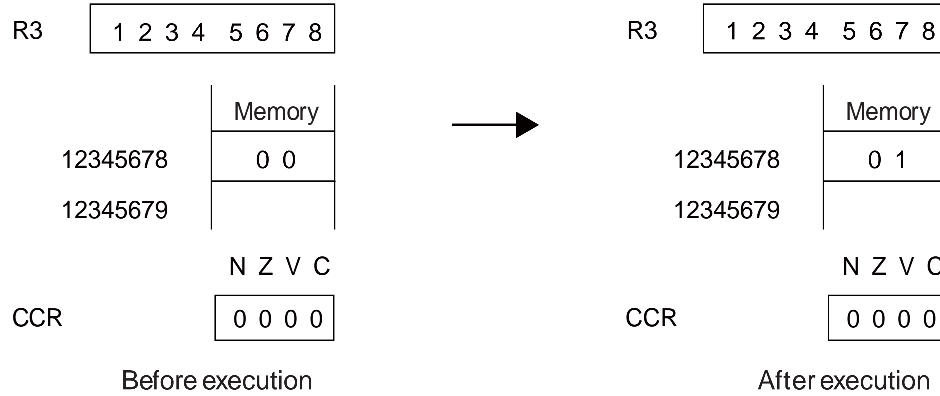
1+2a cycles

- Instruction Format



- Execution Example

BEORL #1,@R3 ; Bit pattern of the instruction: 1001 1000 0001 0011



FR81 Family

7.23 BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory)

Takes the logical OR of the 4-bit immediate data and the higher 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

● Assembler Format

BORH #u4,@Ri

● Operation

(Ri) | {u4 << 4} → (Ri) [Operation uses higher 4 bits only]

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

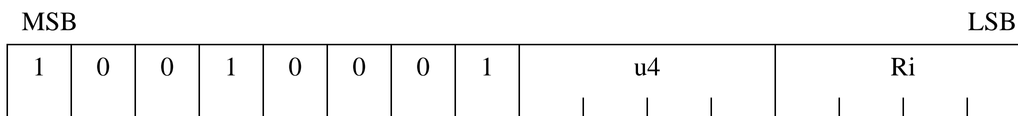
● Classification

Bit Operation instruction, Read/Modify/Write type instruction

● Execution Cycles

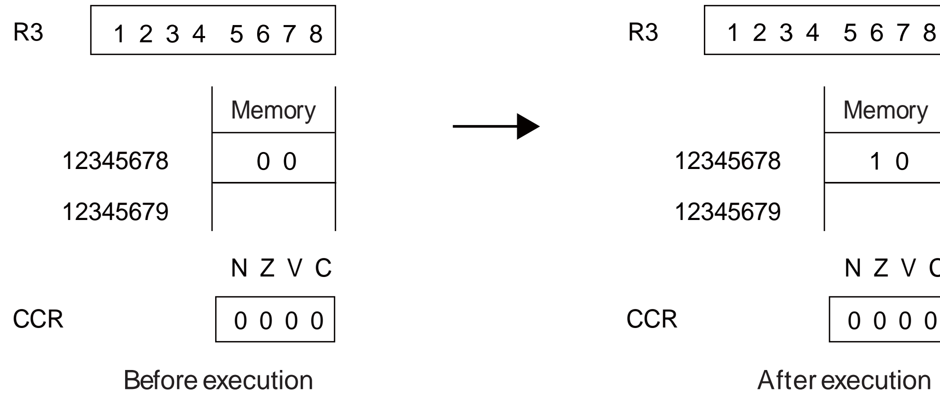
1+2a cycles

● Instruction Format



● Execution Example

BORH #1,@R3 ; Bit pattern of the instruction: 1001 0001 0001 0011



FR81 Family

7.24 BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory)

Takes the logical OR of the 4-bit immediate data and the lower 4 bits of byte data at memory address Ri, stores the results to the memory address corresponding to Ri.

● Assembler Format

BORL #u4,@Ri

● Operation

(Ri) | u4 → (Ri) [Operation uses lower 4 bits only]

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

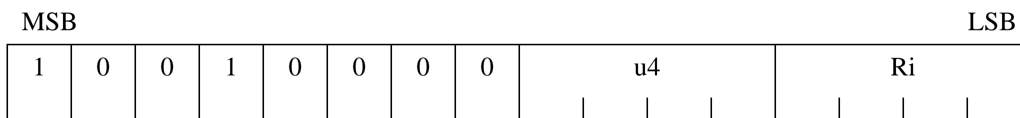
● Classification

Bit Operation instruction, Read/Modify/Write type instruction

● Execution Cycles

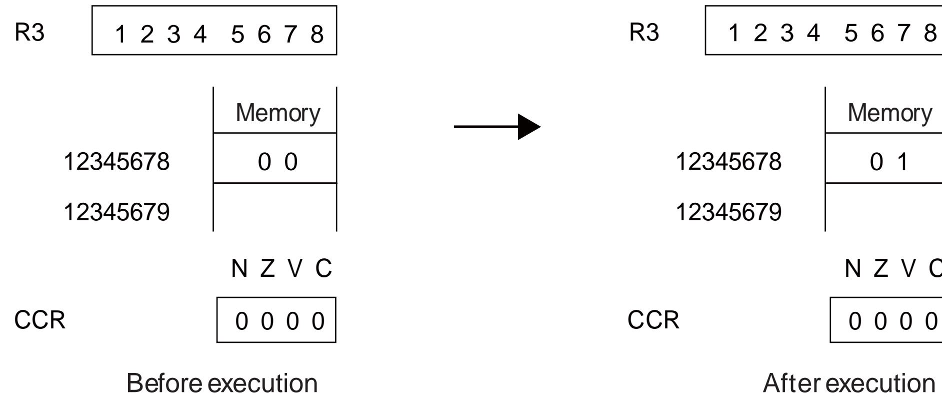
1+2a cycles

● Instruction Format



● Execution Example

BORL #1,@R3 ; Bit pattern of the instruction: 1001 0000 0001 0011



7.25 BTSTH (Test Higher 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the higher 4 bits of byte data at memory address Ri places the results in the condition code register (CCR).

- Assembler Format

BTSTH #u4,@Ri

- Operation

(Ri) & {u4 << 4} [Test uses higher 4 bits only]

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB(bit7) is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

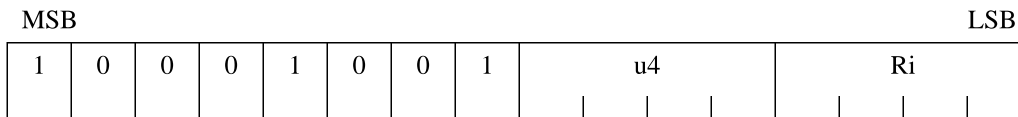
- Classification

Bit Operation instruction

- Execution Cycles

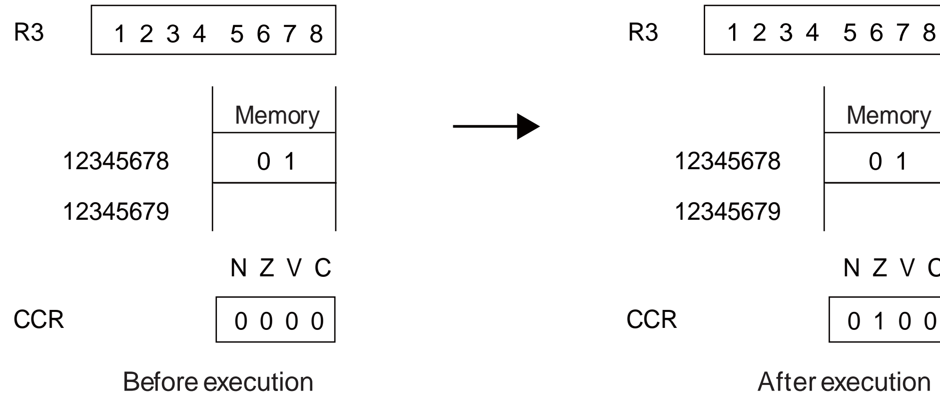
2+a cycles

- Instruction Format



● Execution Example

BTSTH #1,@R3 ; Bit pattern of the instruction: 1000 1001 0001 0011



7.26 BTSTL (Test Lower 4bit of Byte Data in Memory)

Takes the logical AND of the 4-bit immediate data and the lower 4 bits of byte data at memory address Ri, places the results in the flag of the condition code register.

● Assembler Format

BTSTL #u4,@Ri

● Operation

(Ri) & u4 [Test uses lower 4 bits only]

● Flag Change

N	Z	V	C
0	C	-	-

N: Cleared.

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

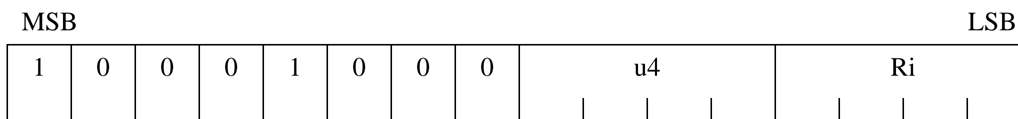
● Classification

Bit Operation instruction

● Execution Cycles

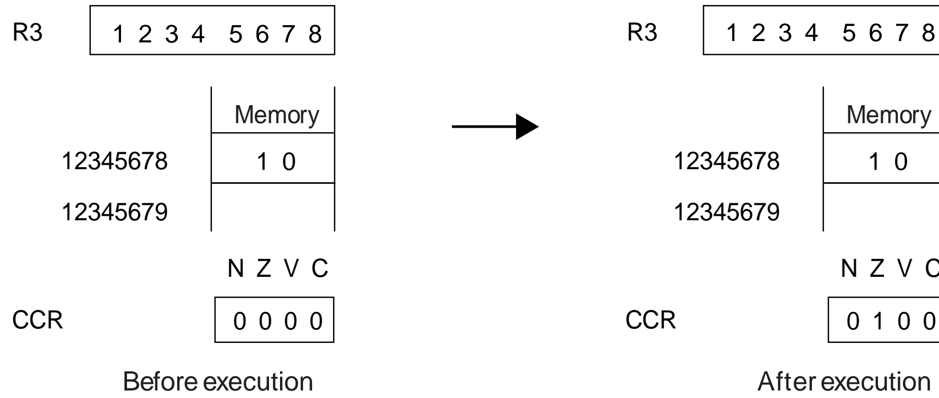
2+a cycles

● Instruction Format



● Execution Example

BTSTL #1,@R3 ; Bit pattern of the instruction: 1000 1000 0001 0011



FR81 Family

7.27 CALL (Call Subroutine)

This is a branching instruction without a delay slot. After storing the address of the next instruction in the return pointer (RP), branch to the address indicated by label12 relative to the value of the program counter (PC). When calculating the address, double the value of rel11 as a signed extension.

- Assembler Format

CALL label12

- Operation

$PC + 2 \rightarrow RP$

$PC + 2 + exts(rel11 \times 2) \rightarrow PC$

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Non-delayed branching instruction

- Execution Cycles

2 cycles

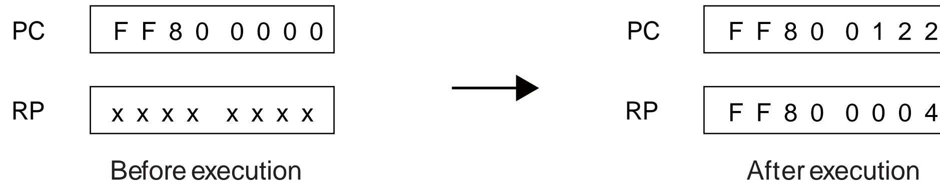
- Instruction Format

MSB (left) … LSB (right)

1	1	0	1	0	rel11

● Execution Example

```
CALL   label          ; Bit pattern of the instruction: 1101 0000 1001 0000
. . .
label:                ; CALL instruction address + 122H
```



FR81 Family

7.28 CALL (Call Subroutine)

This is a branching instruction without a delay slot. After saving the address of the next instruction in the return pointer (RP), a branch to the address indicated by Ri occurs.

● Assembler Format

CALL @Ri

● Operation

PC + 2 → RP

Ri → PC

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

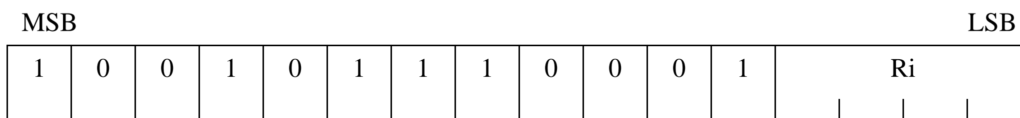
● Classification

Non-delayed branching instruction

● Execution Cycles

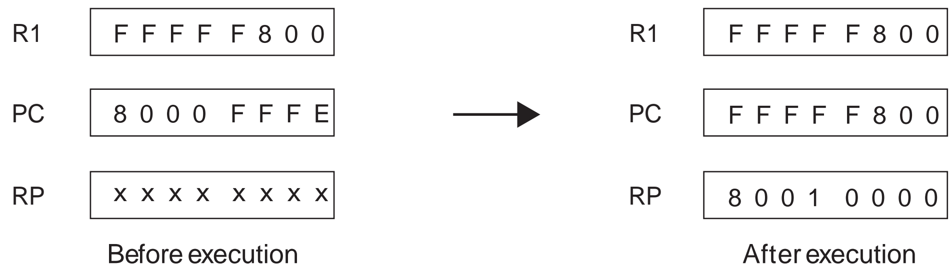
2 cycles

● Instruction Format



- Execution Example

```
CALL   @R1          ; Bit pattern of the instruction: 1001 0111 0001 0001
```



FR81 Family

7.29 CALL:D (Call Subroutine)

This is a branching instruction with a delay slot. After saving the address of the next instruction after the delay slot to the return pointer (RP), branch to the address indicated by label12 relative to the value of the program counter (PC). When calculating the address, double the value of rel11 as a signed extension.

● Assembler Format

CALL:D label12

● Operation

$PC + 4 \rightarrow RP$

$PC + 2 + exts(rel11 \times 2) \rightarrow PC$

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

Delayed branching instruction

● Execution Cycles

1 cycle

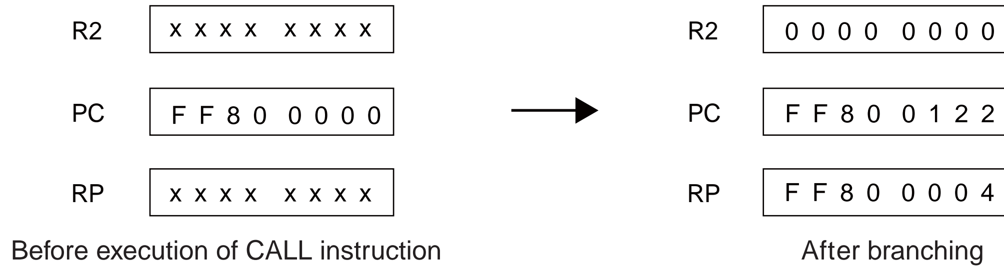
● Instruction Format

MSB LSB

1	1	0	1	1	rel11

● Execution Example

```
CALL:D  label     ; Bit pattern of the instruction: 1101 1000 1001 0000
LDI:8   #0, R2    ; Instruction placed in delay slot
…
label:            ; CALL instruction address + 122H
```



The instruction placed in delay slot is executed before execution of the branch destination instruction. The value R2 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.

FR81 Family

7.30 CALL:D (Call Subroutine)

This is a branching instruction with a delay slot. After saving the address of the next instruction after the delay slot to the return pointer (RP), it branches to the address indicated by Ri.

- Assembler Format

CALL:D @Ri

- Operation

PC + 4 → RP

Ri → PC

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Delayed branching instruction

- Execution Cycles

1 cycle

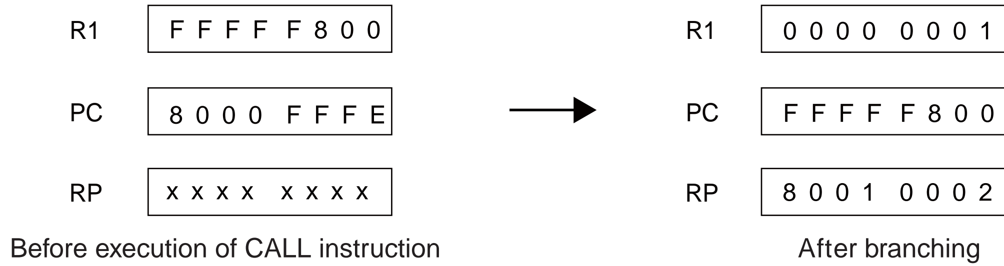
- Instruction Format

MSB ... LSB

1	0	0	1	1	1	1	1	0	0	0	1	Ri			

● Execution Example

```
CALL:D   @R1      ; Bit pattern of the instruction: 1001 1111 0001 0001
LDI:8   #1, R1    ; Instruction placed in delay slot
```



The instruction placed in delay slot is executed before execution of the branch destination instruction. The value R2 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.

FR81 Family

7.31 CMP (Compare Immediate Data and Destination Register)

Subtracts the result of the higher 28 bits of 4-bit immediate data with zero extension from the word data in Ri, sets results in the flag of condition code register (CCR).

- Assembler Format

CMP #i4, Ri

- Operation

Ri - extu(i4)

- Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

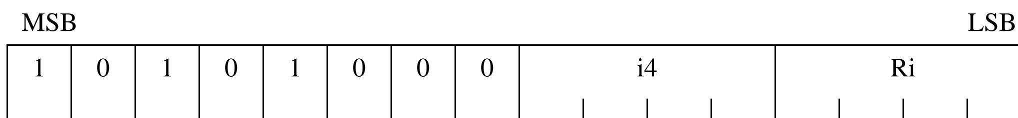
- Classification

Compare instruction, Instruction with delay slot

- Execution Cycles

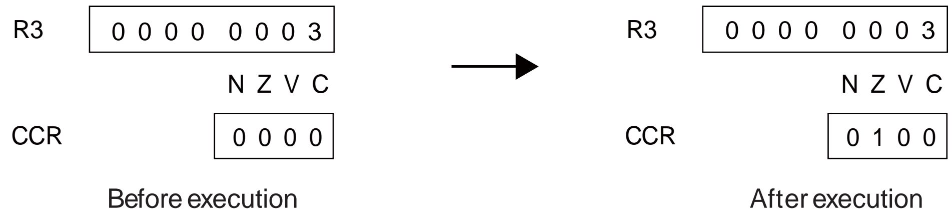
1 cycle

- Instruction Format



- Execution Example

CMP #3, R3 ; Bit pattern of the instruction: 1010 1000 0011 0011



FR81 Family

7.32 CMP (Compare Word Data in Source Register and Destination Register)

Subtracts the word data in Rj from the word data in Ri, sets results in the flag of condition code register (CCR).

● Assembler Format

CMP Rj, Ri

● Operation

Ri - Rj

● Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

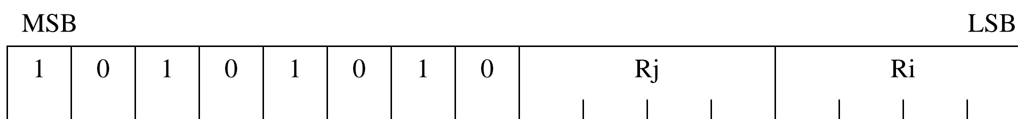
● Classification

Compare instruction, Instruction with delay slot

● Execution Cycles

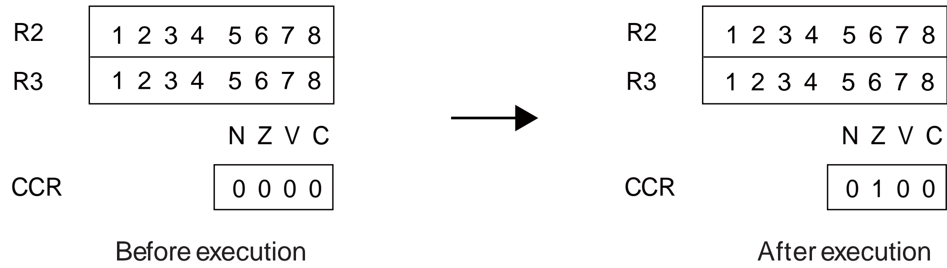
1 cycle

● Instruction Format



- Execution Example

```
CMP   R2, R3          ; Bit pattern of the instruction: 1010 1010 0010 0011
```



7.33 CMP2 (Compare Immediate Data and Destination Register)

Subtracts the result of the higher 28 bits of 4-bit immediate (from -16 to -1) data with minus extension from the word data in Ri, sets results in the flag of condition code register (CCR).

● Assembler Format

CMP2 #i4, Ri

● Operation

Ri - extn(i4)

● Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

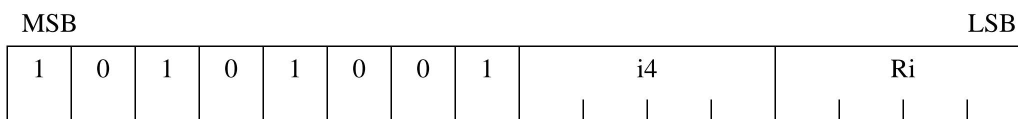
● Classification

Compare instruction, Instruction with delay slot

● Execution Cycles

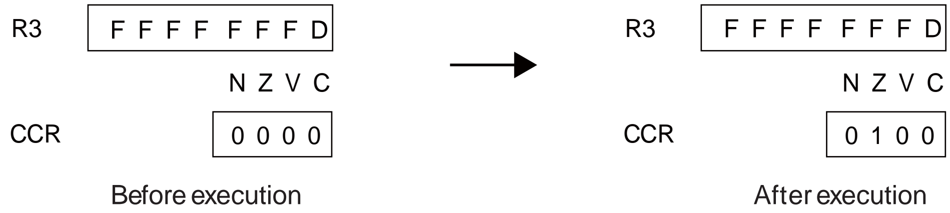
1 cycle

● Instruction Format



- Execution Example

CMP2 #-3, R3 ; Bit pattern of the instruction: 1010 1001 1101 0011



FR81 Family

7.34 DIV0S (Initial Setting Up for Signed Division)

This is a step division instruction. This command issued for signed division in which multiplication division register (MDL) contains the dividend and the Ri the divisor, with the quotient stored in the MDL and the remainder in multiplication division register (MDH).

● Assembler Format

DIV0S Ri

● Operation

MDL[31] → D0

MDL[31] ^ Ri[31] → D1

exts(MDL) → MDH, MDL

The word data in MDL is extended to 64 bits, with the higher word in the MDH and the lower word in the MDL. The value of the sign bit in the MDL and Ri is used to set the D0 and D1 flag bits in the system condition code register (SCR).

● Flag Change

N	Z	V	C
-	-	-	-

D1	D0
C	C

N, Z, V, C: Flags unchanged.

D1: Set when the divisor and dividend signs are different, cleared when equal.

D0: Set when the dividend is negative, cleared when positive.

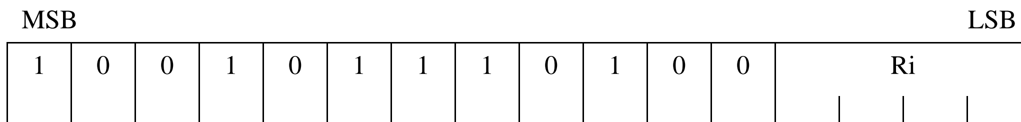
● Classification

Multiply/Divide Instruction

● Execution Cycles

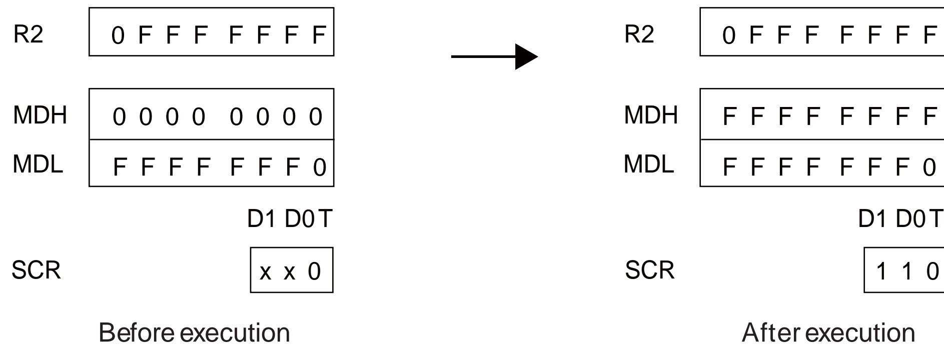
1 cycle

● Instruction Format



● Execution Example

DIV0S R2 ; Bit pattern of the instruction: 1001 0111 0100 0010



FR81 Family

7.35 DIV0U (Initial Setting Up for Unsigned Division)

This is a step division command. This command issued for unsigned division in which multiplication division register (MDL) contains the dividend and the Ri the divisor, with the quotient stored in the MDL and the remainder in multiplication division register (MDH).

● Assembler Format

DIV0U Ri

● Operation

0 → D0

0 → D1

0 → MDH

The MDH and bits D0 and D1 from system condition code register (SCR) are cleared to "0".

● Flag Change

N	Z	V	C
-	-	-	-

D1	D0
0	0

N, Z, V, C: Flags unchanged.

D1,D0: Cleared.

● Classification

Multiply/Divide Instruction

● Execution Cycles

1 cycle

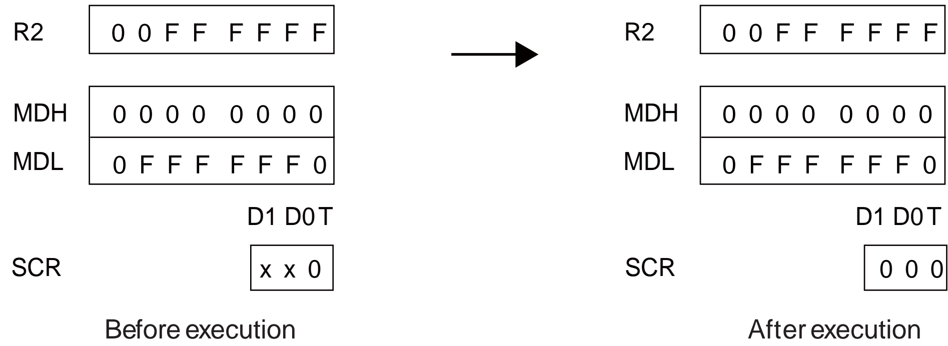
● Instruction Format

MSB ... LSB

1	0	0	1	0	1	1	1	0	1	0	1	Ri

● Execution Example

DIV0U R2 ; Bit pattern of the instruction: 1001 0111 0101 0010



FR81 Family

7.36 DIV1 (Main Process of Division)

This is a step division instruction used for unsigned division.

● Assembler Format

DIV1 Ri

● Operation

```
{MDH, MDL} <<= 1    /* 1 bit left shift */
if (D1==1) {
      MDH + Ri → temp
}
else {
      MDH - Ri → temp
}
if ({D0 ^ D1 ^ C} == 0) {
      temp → MDH
      1 → MDL[0]
}
```

● Flag Change

N	Z	V	C
-	C	-	C

N, V: Unchanged.

Z: Set when the result of step division is zero, cleared otherwise. Set according to remainder of division results, not according to quotient.

C: Set when the operation result of step division involves a carry operation, cleared otherwise.

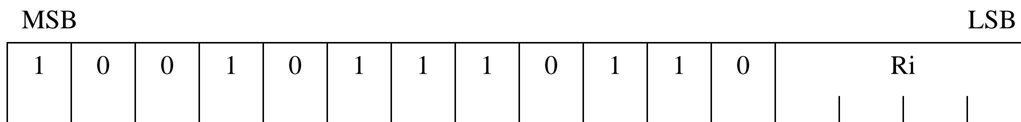
● Classification

Multiply/Divide Instruction

● Execution Cycles

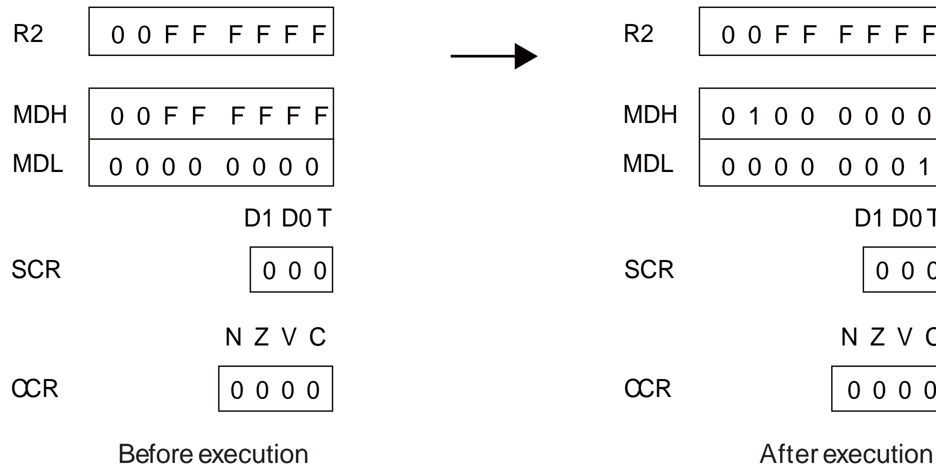
1 cycle

● Instruction Format



● Execution Example

DIV1 R2 ; Bit pattern of the instruction: 1001 0111 0110 0010



FR81 Family

7.37 DIV2 (Correction When Remain is zero)

This is a step division instruction used for signed division.

● Assembler Format

DIV2 Ri

● Operation

```
if (D1==1) {
    MDH + Ri → temp
}
else {
    MDH - Ri → temp
}
if (Z==1) {
    0 → MDH
}
```

● Flag Change

N	Z	V	C
-	C	-	C

N, V: Unchanged.

Z: Set when the result of step division is zero, cleared otherwise. Set according to remainder of division results, not according to quotient.

C: Set when the operation result of step division involves a carry or borrow operation, cleared otherwise.

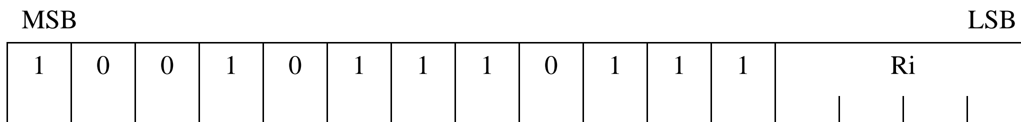
● Classification

Multiply/Divide Instruction

● Execution Cycles

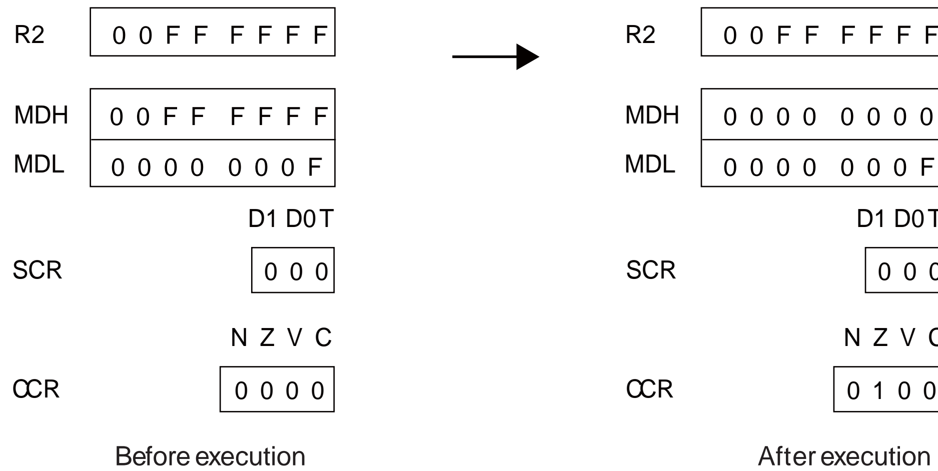
c cycle

- Instruction Format



- Execution Example

DIV2 R2 ; Bit pattern of the instruction: 1001 0111 0111 0010



7.38 DIV3 (Correction When Remain is zero)

This is a step division instruction used for signed division.

- Assembler Format

DIV3

- Operation

```
if (Z==1) {
    MDL + 1 → MDL
}
```

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Multiply/Divide Instruction

- Execution Cycles

1 cycle

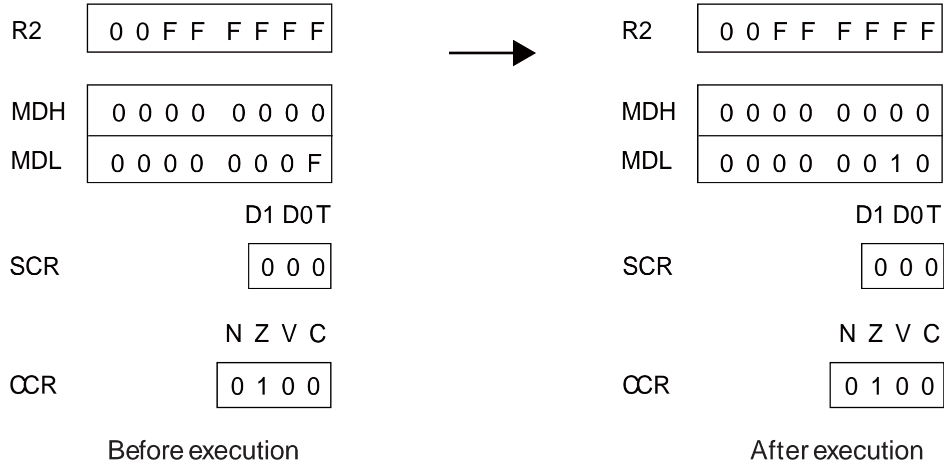
- Instruction Format

MSB ... LSB

1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0

- Execution Example

```
DIV3                    ; Bit pattern of the instruction: 1001 1111 0110 0000
```



FR81 Family

7.39 DIV4S (Correction Answer for Signed Division)

This is a step division instruction used for signed division.

- Assembler Format

DIV4S

- Operation

```
if (D1==1) {
    0 - MDL → MDL
}
```

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Multiply/Divide Instruction

- Execution Cycles

1 cycle

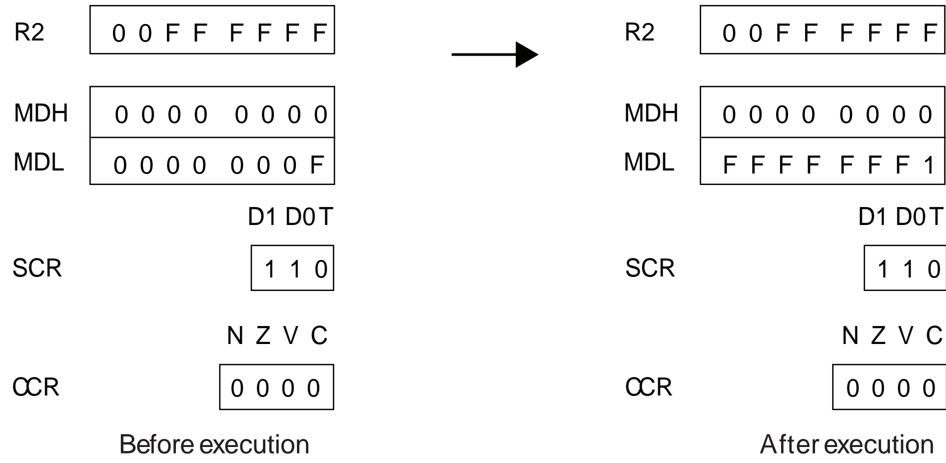
- Instruction Format

MSB ... LSB

1	0	0	1	1	1	1	1	0	1	1	1	0	0	0	0

- Execution Example

DIV4S ; Bit pattern of the instruction: 1001 1111 0111 0000



FR81 Family

7.40 DMOV (Move Word Data from Direct Address to Register)

Transfers, to R13 the word data at the direct address corresponding to 4 times the value of dir8. The value of dir8 × 4 is specified as dir10.

- Assembler Format

DMOV @dir10, R13

- Operation

(dir8 × 4) → R13

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

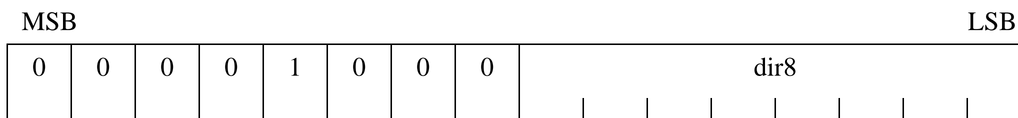
- Classification

Direct Addressing Instructions, Instruction with delay slot

- Execution Cycles

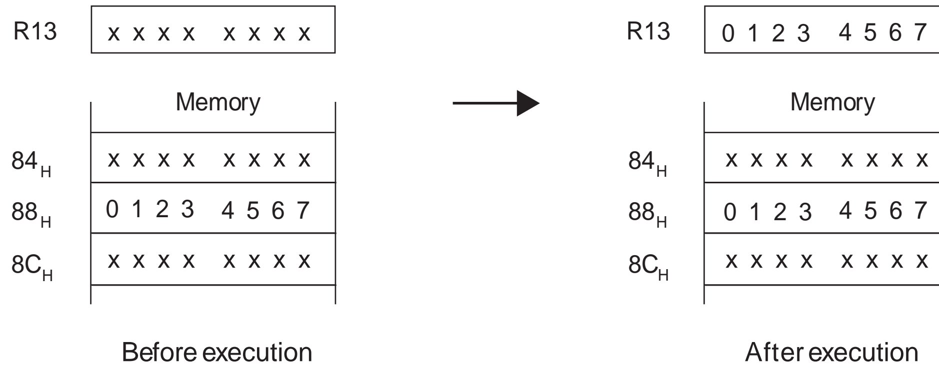
b cycle

- Instruction Format



● Execution Example

DMOV @88H, R13 ; Bit pattern of the instruction: 0000 1000 0010 0010



FR81 Family

7.41 DMOV (Move Word Data from Register to Direct Address)

Transfers word data in R13 to the direct address corresponding to 4 times the value of dir8. The value of dir8 × 4 is specified as dir10.

- Assembler Format

DMOV R13,@dir10

- Operation

R13 → (dir8 × 4)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

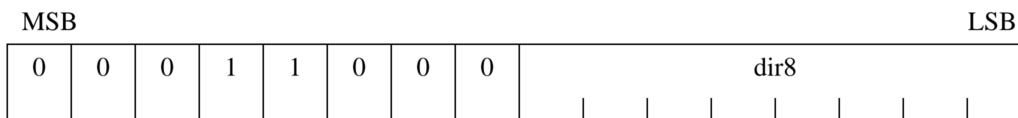
- Classification

Direct Addressing Instructions, Instruction with delay slot

- Execution Cycles

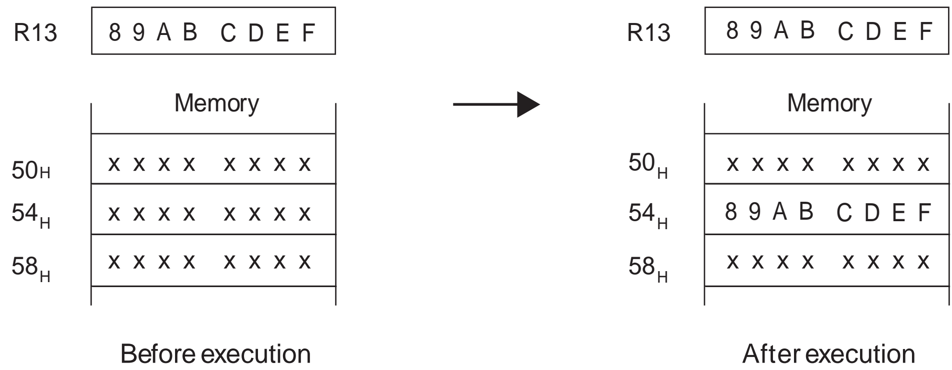
a cycle

- Instruction Format



- Execution Example

DMOV R13,@54H ; Bit pattern of the instruction: 0001 1000 0001 0101



FR81 Family

7.42 DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address)

Transfers the word data at the direct address corresponding to 4 times the value of dir8 to the address indicated in R13. After the data transfer, it increments the value of R13 by 4. The value of dir8 × 4 is specified as dir10.

- Assembler Format

DMOV @dir10,@R13+

- Operation

$(dir8 \times 4) \rightarrow (R13)$

$R13 + 4 \rightarrow R13$

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

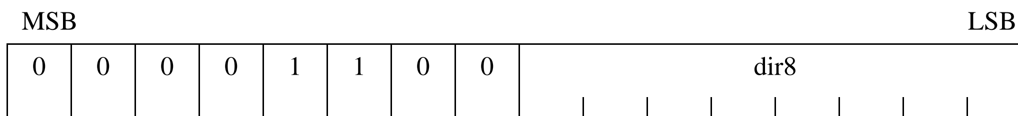
- Classification

Direct Addressing Instructions

- Execution Cycles

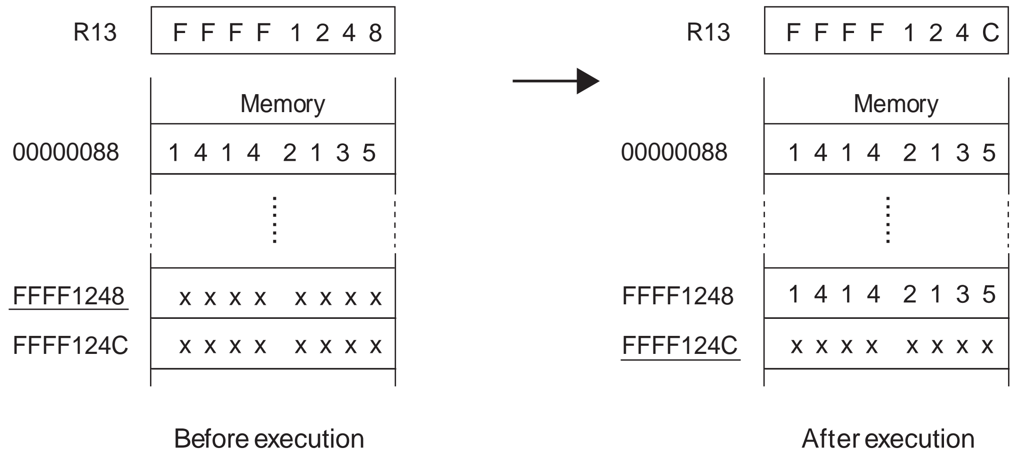
1+2a cycles

- Instruction Format



● Execution Example

DMOV @88H,@R13+ ; Bit pattern of the instruction: 0000 1100 0010 0010



7.43 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the word data at the address indicated in R13 to the direct address corresponding to 4 times the value dir8. After the data transfer, it increments the value of R13 by 4. The value of dir8 × 4 is specified as dir10.

- Assembler Format

DMOV @R13+,@dir10

- Operation

(R13) → (dir8 × 4)

R13 + 4 → R13

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

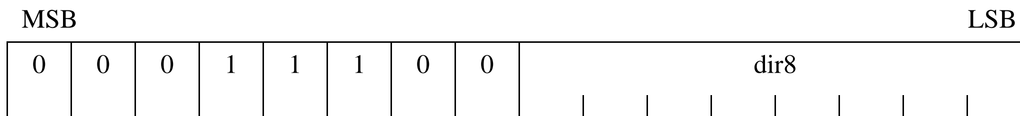
- Classification

Direct Addressing Instructions

- Execution Cycles

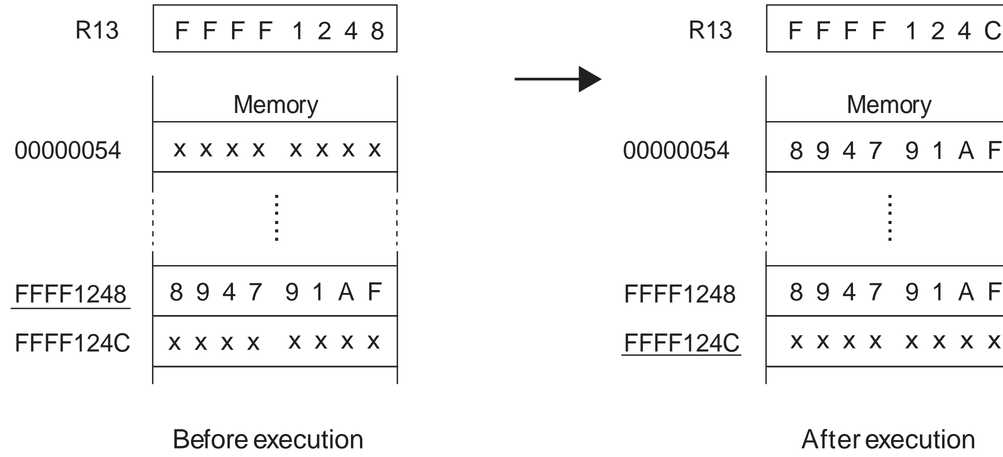
1+2a cycles

- Instruction Format



● Execution Example

DMOV @R13+,@54H ; Bit pattern of the instruction: 0001 1100 0001 0101



FR81 Family

7.44 DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address)

Decrements the value of R15 by 4, then transfers the word data at the direct address corresponding to 4 times the value of dir8 to the address indicated in R15. The value of dir8 × 4 is specified as dir10.

- Assembler Format

DMOV @dir10,@-R15

- Operation

R15 - 4 → R15
(dir8 × 4) → (R15)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

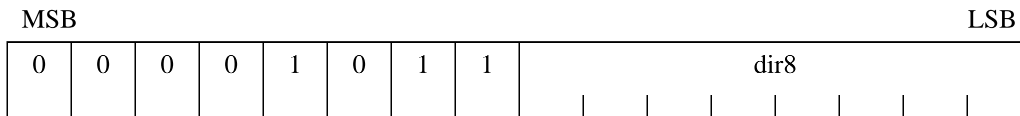
- Classification

Direct Addressing Instructions

- Execution Cycles

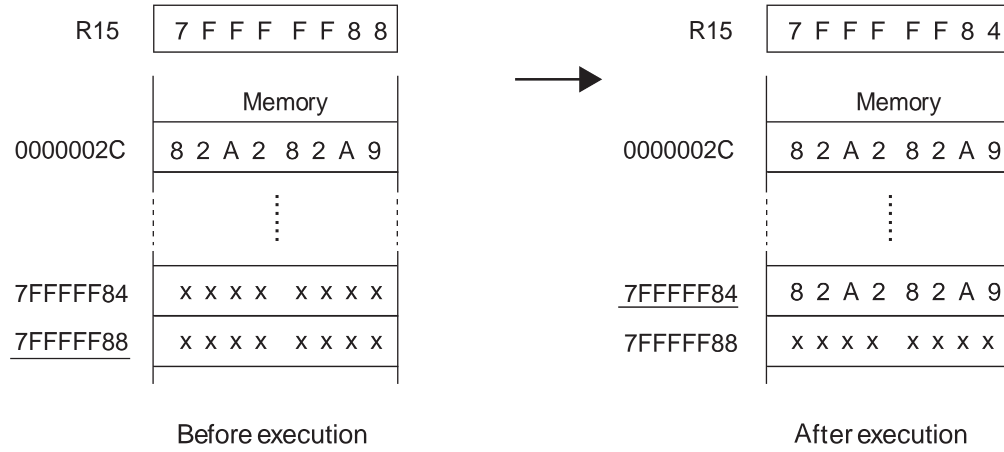
1+2a cycles

- Instruction Format



● Execution Example

DMOV @2CH,@-R15 ; Bit pattern of the instruction: 0000 1011 0000 1011



7.45 DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address)

Transfers the word data at the address indicated in R15 to the direct address corresponding to 4 times the value dir8. After the data transfer, it increments the value of R15 by 4. The value of dir8 × 4 is specified as dir10.

- Assembler Format

DMOV @R15+,@dir10

- Operation

(R15) $\rightarrow$ (dir8 $\times$ 4)

R15 + 4 $\rightarrow$ R15

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

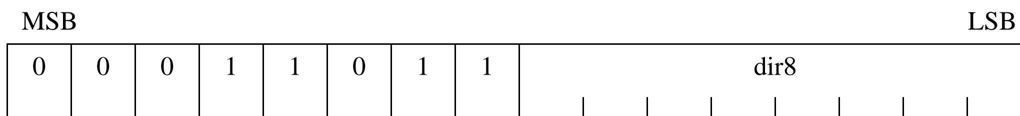
- Classification

Direct Addressing Instructions

- Execution Cycles

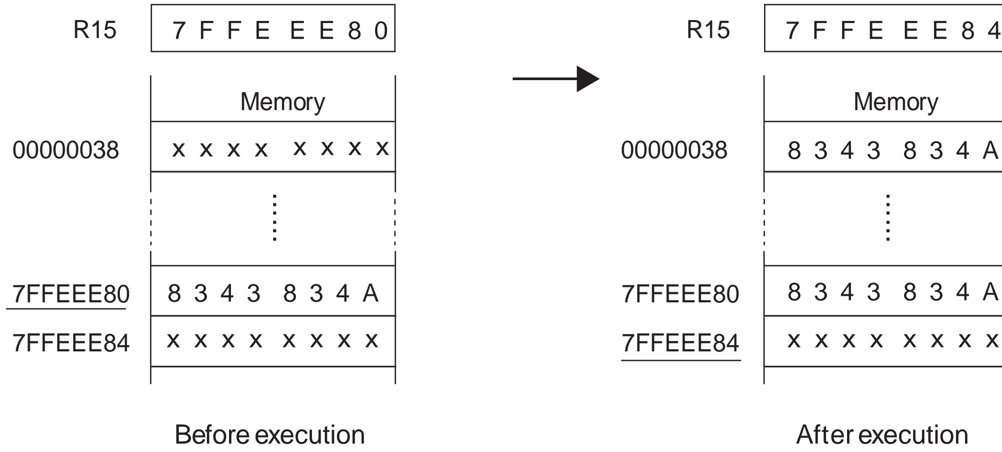
1+2a cycles

- Instruction Format



● Execution Example

DMOV @R15+,@38H ; Bit pattern of the instruction: 0001 1011 0000 1110



FR81 Family

7.46 DMOVB (Move Byte Data from Direct Address to Register)

Transfers the byte data at the address indicated by the value dir8 to R13. Uses zeros to extend the higher 24 bits of data.

- Assembler Format

DMOVB @dir8, R13

- Operation

(dir8) → R13

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

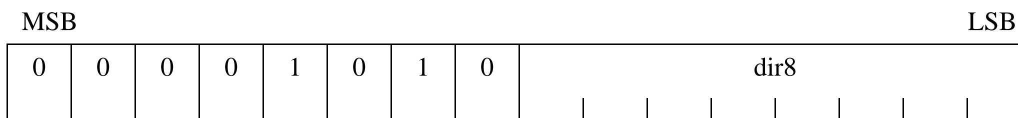
- Classification

Direct Addressing Instructions, Instruction with delay slot

- Execution Cycles

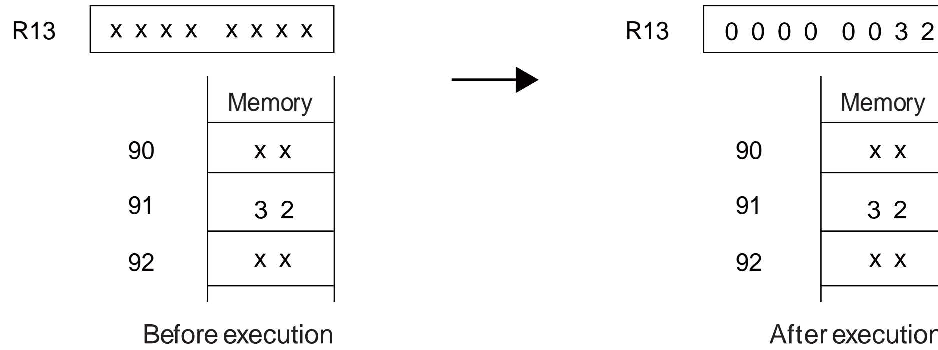
b cycle

- Instruction Format



● Execution Example

DMOVB @91H, R13 ; Bit pattern of the instruction: 0000 1010 1001 0001



FR81 Family

7.47 DMOVB (Move Byte Data from Register to Direct Address)

Transfers the byte data from R13 to the direct address indicated by the value dir8.

● Assembler Format

DMOVB R13,@dir8

● Operation

R13 → (dir8)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

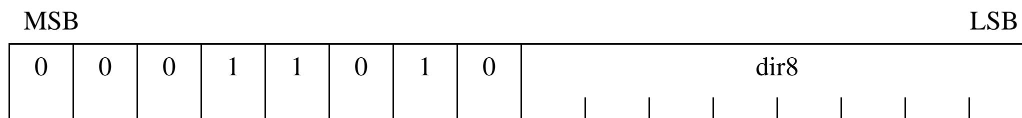
● Classification

Direct Addressing Instructions, Instruction with delay slot

● Execution Cycles

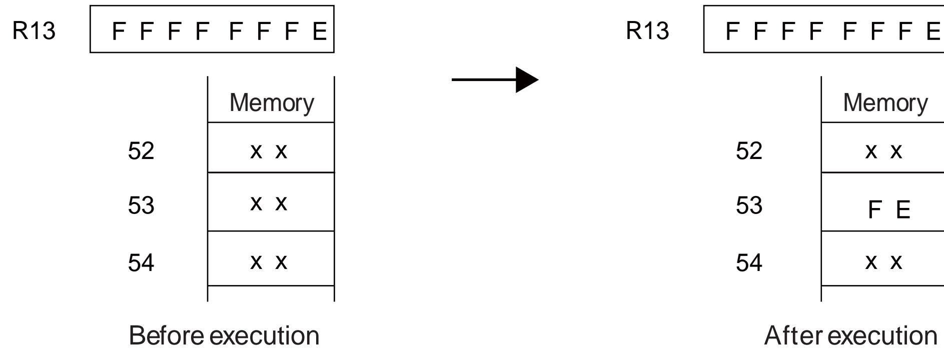
a cycle

● Instruction Format



- Execution Example

DMOVB R13,@53H ; Bit pattern of the instruction: 0001 1010 0101 0011



FR81 Family

7.48 DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address)

Moves the byte data at the direct address indicated by the value dir8 to the address indicated by R13. After the data transfer, it increments the value of R13 by 1.

- Assembler Format

DMOVB @dir8,@R13+

- Operation

(dir8) → (R13)

R13 + 1 → R13

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

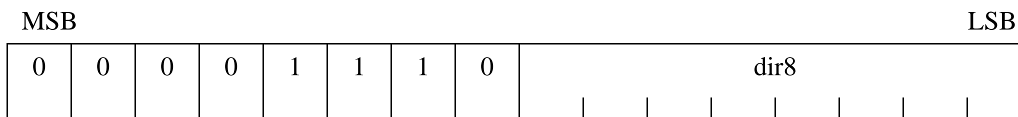
- Classification

Direct Addressing Instructions

- Execution Cycles

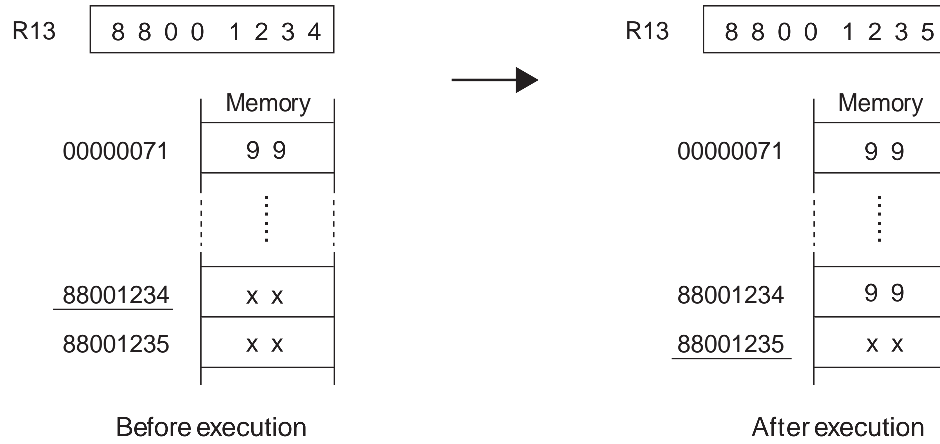
1+2a cycles

- Instruction Format



● Execution Example

DMOVB @71H,@R13+ ; Bit pattern of the instruction: 0000 1110 0111 0001



FR81 Family

7.49 DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address)

Transfers the byte data at the address indicated by R13 to the direct address indicated by the value dir8. After the data transfer, it increments the value of R13 by 1.

● Assembler Format

DMOVB @R13+,@dir8

● Operation

(R13) → (dir8)

R13 + 1 → R13

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

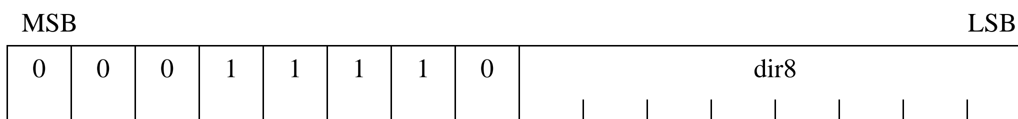
● Classification

Direct Addressing Instructions

● Execution Cycles

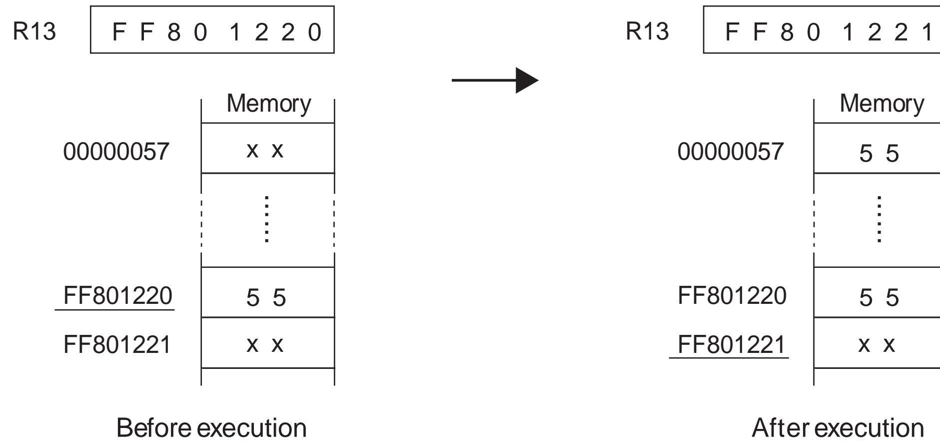
1+2a cycles

● Instruction Format



● Execution Example

DMOVB @R13+,@57H ; Bit pattern of the instruction: 0001 1110 0101 0111



FR81 Family

7.50 DMOVH (Move Halfword Data from Direct Address to Register)

Transfers the half-word data at the direct address corresponding to 2 times the value dir8 to R13. Uses zeros to extend the higher 16 bits of data. The value of dir8 × 2 is specified as dir9.

● Assembler Format

DMOVH @dir9, R13

● Operation

$(dir8 \times 2) \rightarrow R13$

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

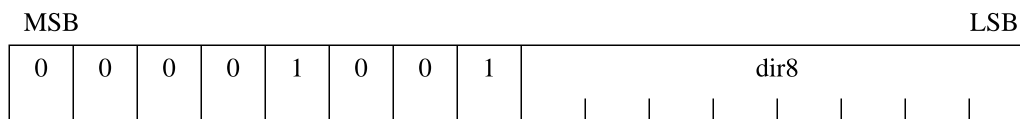
● Classification

Direct Addressing Instructions, Instruction with delay slot

● Execution Cycles

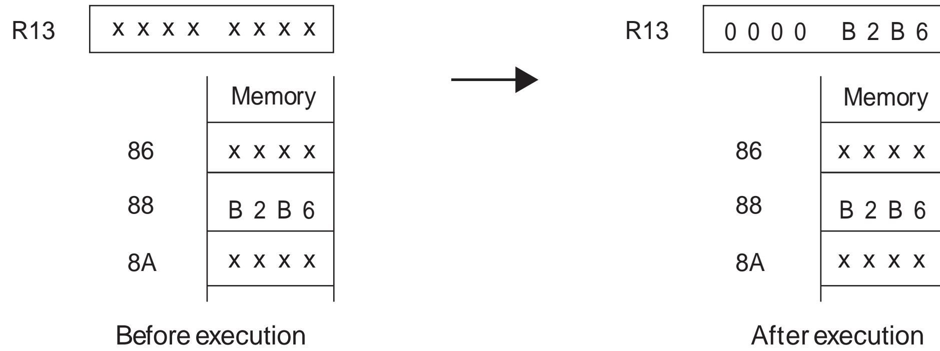
b cycle

● Instruction Format



- Execution Example

```
DMOVH   @88H, R13       ; Bit pattern of the instruction: 0000 1001 0100 0100
```



FR81 Family

7.51 DMOVH (Move Halfword Data from Register to Direct Address)

Transfers the half-word data from R13 to the direct address corresponding to 2 times the value dir8. The value of dir8 × 2 is specified as dir9.

- Assembler Format

DMOVH R13,@dir9

- Operation

R13 → (dir8 × 2)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

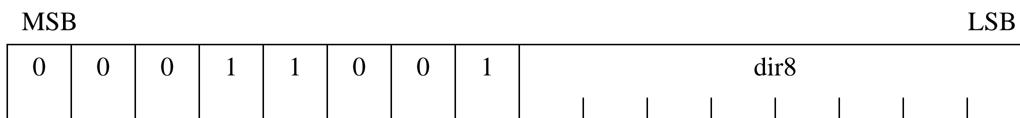
- Classification

Direct Addressing Instructions, Instruction with delay slot

- Execution Cycles

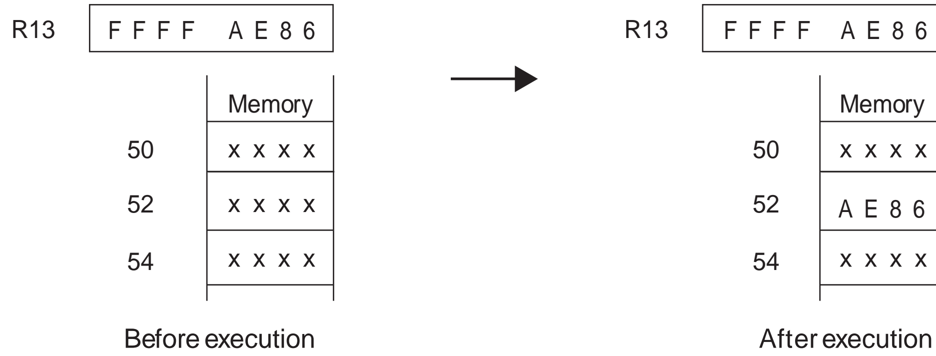
a cycle

- Instruction Format



- Execution Example

DMOVH R13,@52H ; Bit pattern of the instruction: 0001 1001 0010 1001



7.52 DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address)

Transfers the half-word data at the direct address corresponding to 2 times the value dir8 to the address indicated by R13. After the data transfer, it increments the value of R13 by 2. The value of dir8 × 2 is specified as dir9.

- Assembler Format

DMOVH @dir9,@R13+

- Operation

$(dir8 \times 2) \rightarrow (R13)$

$R13 + 2 \rightarrow R13$

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

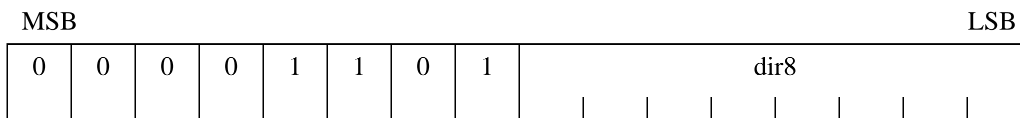
- Classification

Direct Addressing Instructions

- Execution Cycles

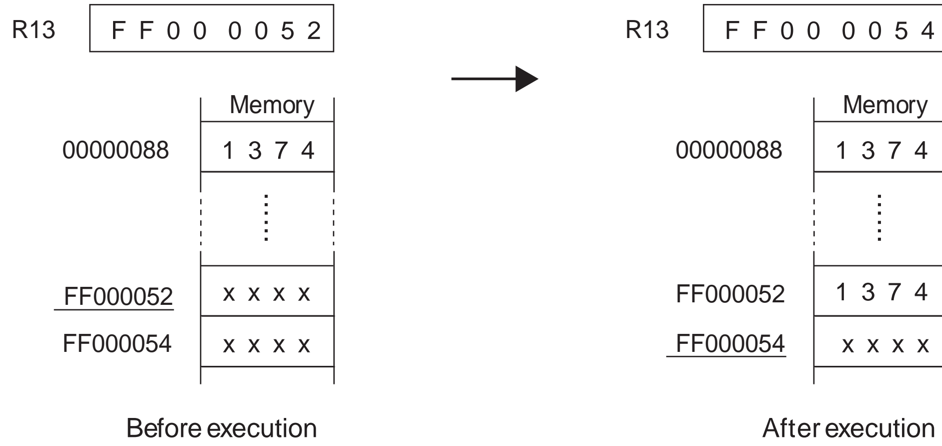
1+2a cycles

- Instruction Format



- Execution Example

DMOVH @88H,@R13+ ; Bit pattern of the instruction: 0000 1101 0100 0100



FR81 Family

7.53 DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address)

Transfers the half-word data at the address indicated by R13 to the direct address corresponding to 2 times the value dir8. After the data transfer, it increments the value of R13 by 2. The value of dir8 × 2 is specified as dir9.

- Assembler Format

DMOVH @R13+,@dir9

- Operation

(R13) → (dir8 × 2)

R13 + 2 → R13

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

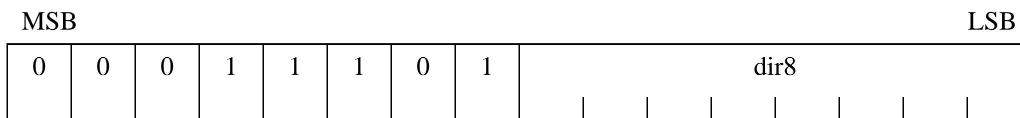
- Classification

Direct Addressing Instructions

- Execution Cycles

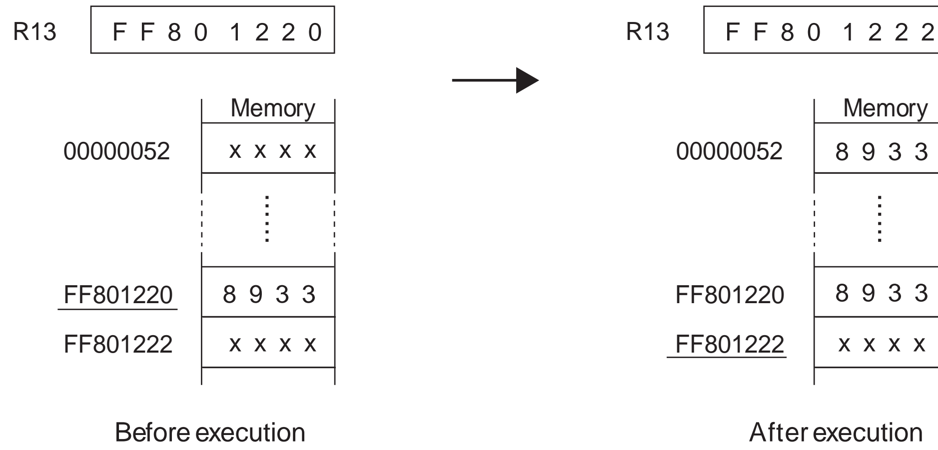
1+2a cycles

- Instruction Format



● Execution Example

DMOVH @R13+,@52H ; Bit pattern of the instruction: 0001 1101 0010 1001



FR81 Family

7.54 ENTER (Enter Function)

This instruction is used for stack frame generation processing for high level languages.The value u8 is calculated as an unsigned value. The value of u8 × 4 is specified as u10.

● Assembler Format

ENTER #u10

● Operation

R14 → (R15-4)

R15 - 4 → R14

R15 - extu(u8 × 4) → R15

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

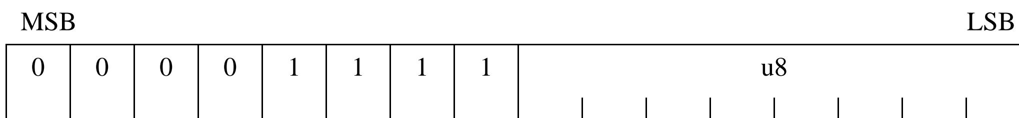
● Classification

Other instructions

● Execution Cycles

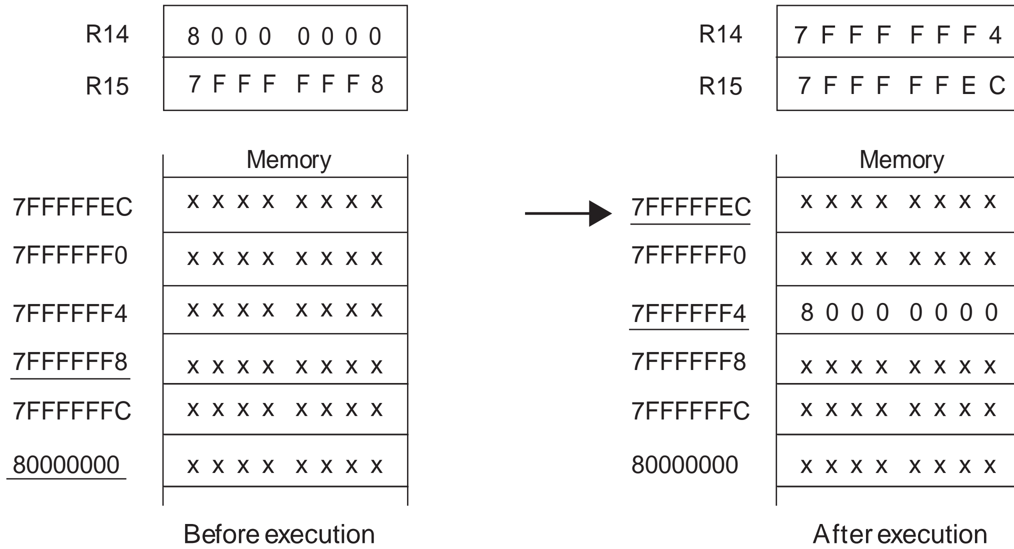
1+a cycles

● Instruction Format



- Execution Example

ENTER #0CH ; Bit pattern of the instruction: 0000 1111 0000 0011



7.55 EOR (Exclusive Or Word Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the word data at memory address Ri and the word data in Rj, stores the results to the memory address corresponding to Ri.

● Assembler Format

EOR Rj,@Ri

● Operation

(Ri) ^ Rj → (Ri)

● Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

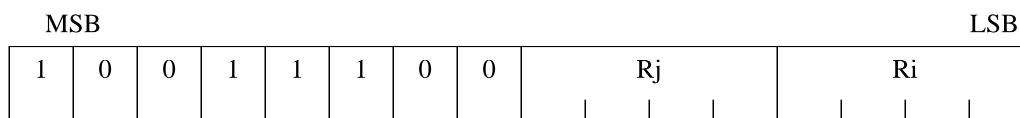
● Classification

Logical Calculation instruction, Read/Modify/Write type instruction

● Execution Cycles

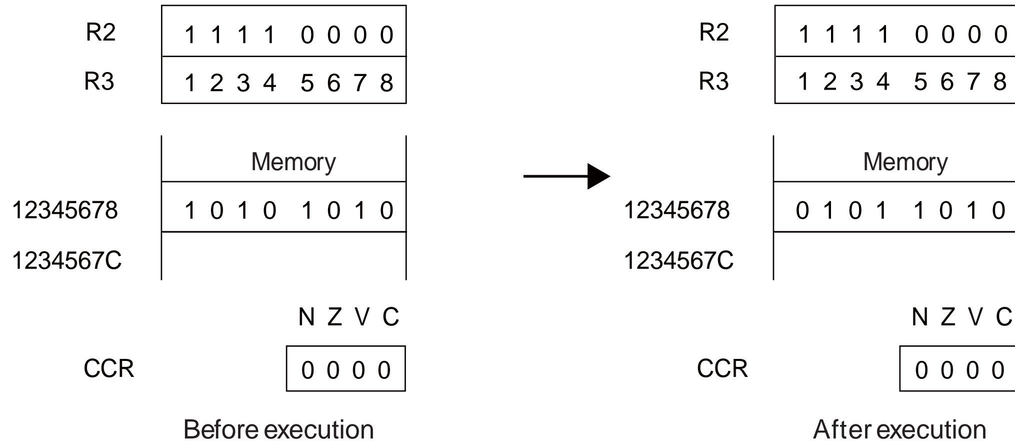
1+2a cycles

● Instruction Format



- Execution Example

EOR R2,@R3 ; Bit pattern of the instruction: 1001 1100 0010 0011



7.56 EOR (Exclusive Or Word Data of Source Register to Destination Register)

Takes the logical exclusive OR of the word data in Ri and the word data in Rj, stores the results to Ri.

- Assembler Format

EOR Rj, Ri

- Operation

Ri ^ Rj → Ri

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

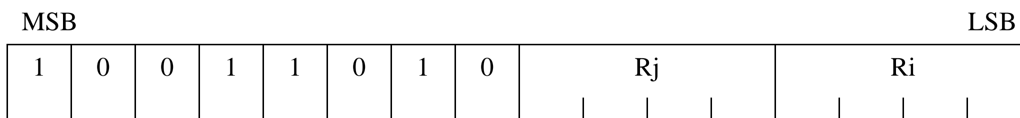
- Classification

Logical Calculation instruction, Instruction with delay slot

- Execution Cycles

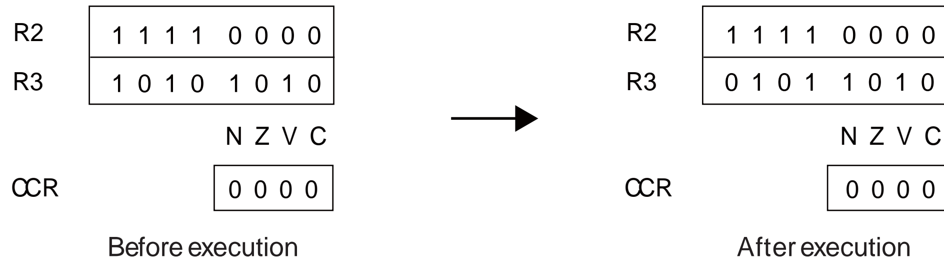
1 cycle

- Instruction Format



- Execution Example

EOR R2, R3 ; Bit pattern of the instruction: 1001 1010 0010 0011



7.57 EORB (Exclusive Or Byte Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the byte data at memory address Ri and the byte datain Rj, stores the results to the memory address corresponding to Ri.

- Assembler Format

EORB Rj,@Ri

- Operation

(Ri) ^ Rj → (Ri)

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

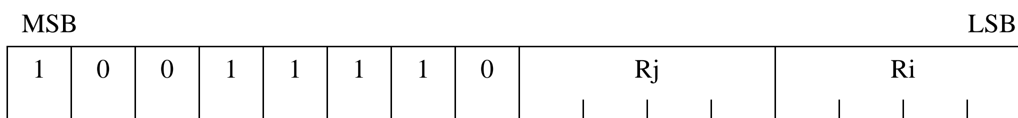
- Classification

Logical Calculation instruction, Read/Modify/Write type instruction

- Execution Cycles

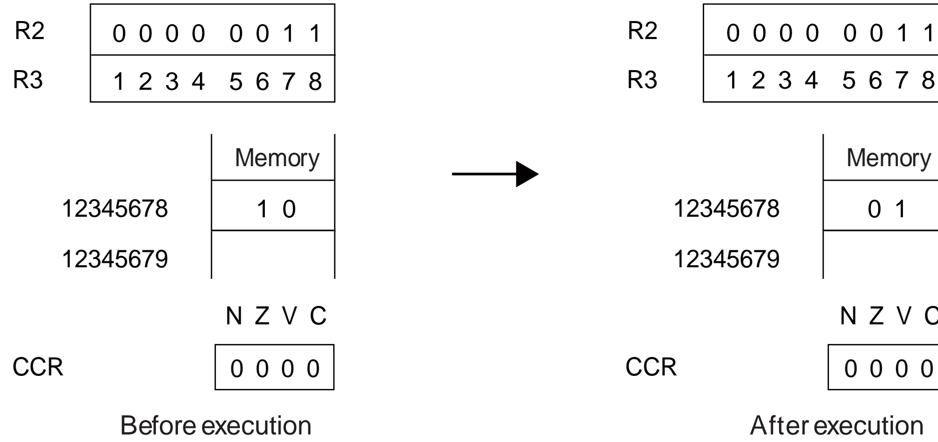
1+2a cycles

- Instruction Format



- Execution Example

EORB R2,@R3 ; Bit pattern of the instruction: 1001 1110 0010 0011



FR81 Family

7.58 EORH (Exclusive Or Halfword Data of Source Register to Data in Memory)

Takes the logical exclusive OR of the half-word data at memory address Ri and the half-word data in Rj, stores the results to the memory address corresponding to Ri.

● Assembler Format

EORH Rj,@Ri

● Operation

(Ri) ^ Rj → (Ri)

● Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB(bit15) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

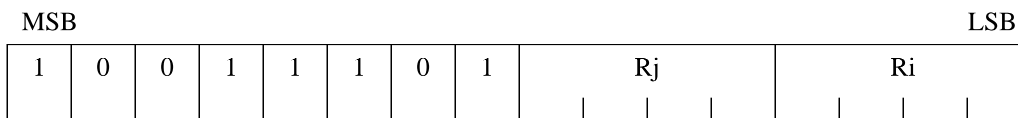
● Classification

Logical Calculation instruction, Read/Modify/Write type instruction

● Execution Cycles

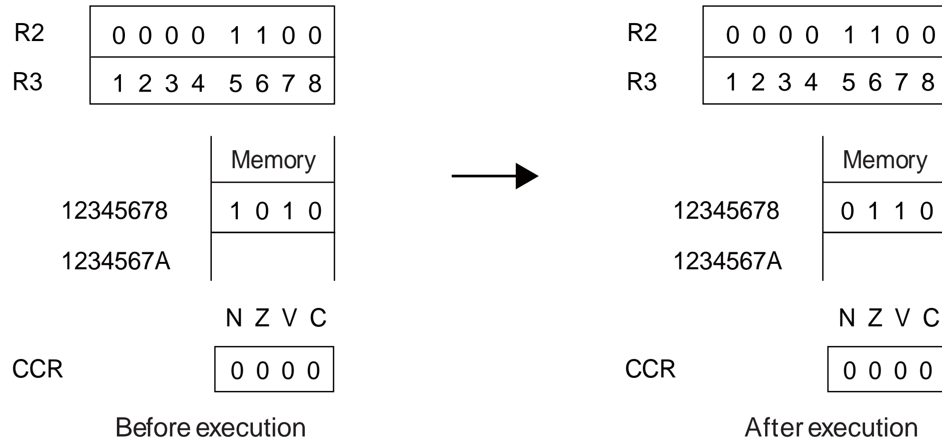
1+2a cycles

● Instruction Format



● Execution Example

EORH R2,@R3 ; Bit pattern of the instruction: 1001 1101 0010 0011



FR81 Family

7.59 EXTSB (Sign Extend from Byte Data to Word Data)

Extends the byte data indicated by Ri to word data as signed binary value.

- Assembler Format

EXTSB Ri

- Operation

exts(Ri[7:0]) → Ri [byte → word]

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

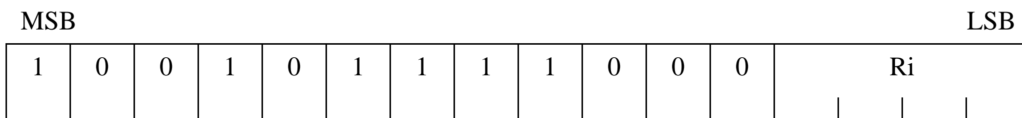
- Classification

Other instructions, Instruction with delay slot

- Execution Cycles

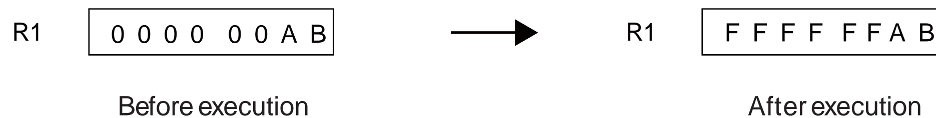
1 cycle

- Instruction Format



● Execution Example

EXTSB R1 ; Bit pattern of the instruction: 1001 0111 1000 0001



FR81 Family

7.60 EXTSH (Sign Extend from Byte Data to Word Data)

Extends the half-word data indicated by Ri to word data as a signed binary value.

● Assembler Format

EXTSH Ri

● Operation

exts(Ri[15:0]) → Ri [half-word → word]

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

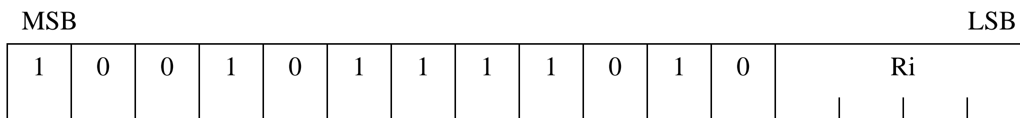
● Classification

Other instructions, Instruction with delay slot

● Execution Cycles

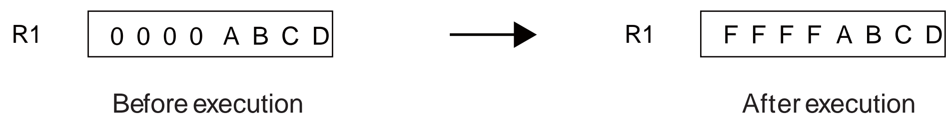
1 cycle

● Instruction Format



● Execution Example

EXTSH R1 ; Bit pattern of the instruction: 1001 0111 1010 0001



FR81 Family

7.61 EXTUB (Unsign Extend from Byte Data to Word Data)

Extends the byte data indicated by Ri to word data as an unsigned binary value.

- Assembler Format

EXTUB Ri

- Operation

extu(Ri[7:0]) → Ri [byte → word]

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

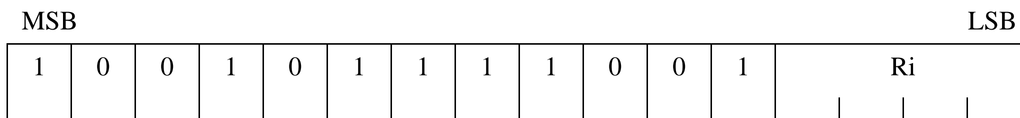
- Classification

Other instructions, Instruction with delay slot

- Execution Cycles

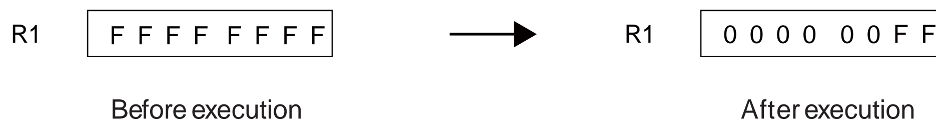
1 cycle

- Instruction Format



● Execution Example

EXTUB R1 ; Bit pattern of the instruction: 1001 0111 1001 0001



FR81 Family

7.62 EXTUH (Unsign Extend from Byte Data to Word Data)

Extends the half-word data indicated by Ri to word data as an unsigned binary value.

● Assembler Format

EXTUH Ri

● Operation

extu(Ri[15:0]) → Ri [half-word → word]

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

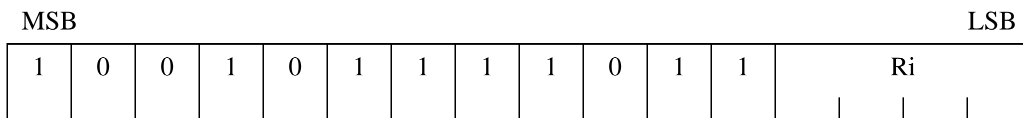
● Classification

Other instructions, Instruction with delay slot

● Execution Cycles

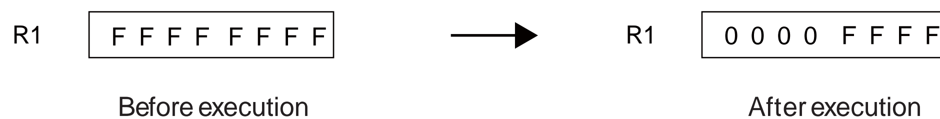
1 cycle

● Instruction Format



- Execution Example

```
EXTUH   R1          ; Bit pattern of the instruction: 1001 0111 1011 0001
```



FR81 Family

7.63 FABSs (Single Precision Floating Point Absolute Value)

Loads the absolute value FRj to FRi.

● Assembler Format

FABSs FRj, FRi

● Operation

| FRj | → FRi

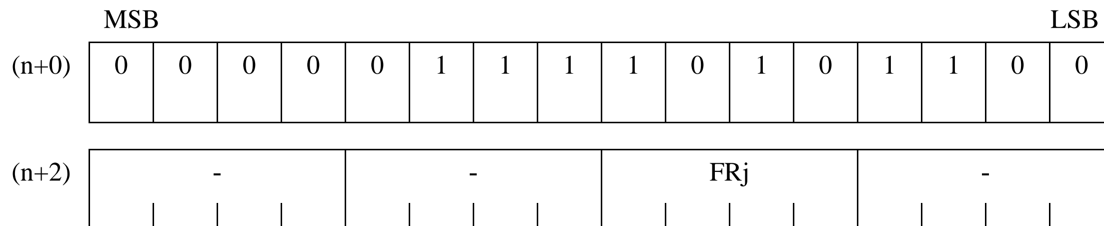
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an interrupt is detected.

7.64 FADDs (Single Precision Floating Point Add)

FRk is added to FRj, and its result is stored in FRi.

- Assembler Format

FADDs FRk, FRj, FRi

- Operation

FRk + FRj → FRi

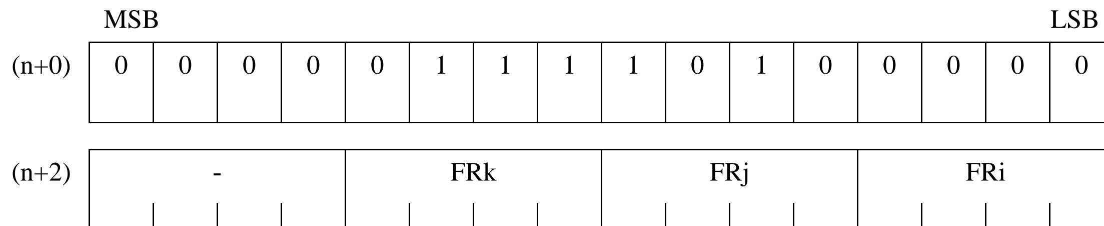
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

FR81 Family

- Calculation result and exception flag

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">FRj</td></tr>
<tr><td>+0</td><td>-0</td><td>+Norm</td><td>-Norm</td><td>+INF</td><td>-INF</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">FRk</td><td>+0</td><td colspan="2">+0/(-)</td><td rowspan="2">+Norm/(X)</td><td rowspan="2">-Norm/(X)</td><td rowspan="4">+INF/(-)</td><td rowspan="4">-INF/(-)</td><td rowspan="6">QNaN/(-)</td><td rowspan="8">QNaN/V</td></tr>
<tr><td>-0</td><td></td><td>-0/(-)</td></tr>
<tr><td>+Norm</td><td colspan="2">+Norm/(X)</td><td>*1</td><td>*2</td></tr>
<tr><td>-Norm</td><td colspan="2">-Norm/(X)</td><td>*2</td><td>*1</td></tr>
<tr><td>+INF</td><td colspan="5"></td><td>QNaN/V</td></tr>
<tr><td>-INF</td><td colspan="4">-INF/(-)</td><td>QNaN/V</td><td>-INF/(-)</td></tr>
<tr><td>QNaN</td><td colspan="7"></td></tr>
<tr><td>SNaN</td><td colspan="7"></td></tr>
</table>

*1: +INF/0,X or -INF/P.X or ± Norm/(X)

*2: ± 0/(-) or ± 0/U or ± Norm/(X)

7.65 FBcc (Floating Point Conditional Branch)

This is a branching instruction without a delay slot. If conditions specified for each instruction are satisfied, control branches to the address indicated by label17 relative to the program counter (PC). The rel16 value is doubled and its sign is extended. If conditions are not satisfied, no branching occurs.

● Assembler Format

FBN		FBL	label17
FBA	label17	FBUGE	label17
FBNE	label17	FBUG	label17
FBE	label17	FBLE	label17
FBLG	label17	FBG	label17
FBUE	label17	FBULE	label17
FBUL	label17	FBU	label17
FBGE	label17	FBO	label17

The FBN instruction has no operand.

● Operation

if (condition) {

$PC + 4 + exts(rel16 \times 2) \rightarrow PC$

}

The branching conditions of each instruction are shown in Table 7.65-1.

Table 7.65-1 Branching conditions of FBcc instruction (1 / 2)

Mnemonic	cc	Contents	conditions (FCC field)
FBN	0000	Branch Never	Always unsatisfied
FBA	1111	Branch Always	Always satisfied
FBNE	0111	Branch Not Equal	L or G or U
FBE	1000	Branch Equal	E
FBLG	0110	Branch Less or Greater	L or G
FBUE	1001	Branch Unordered or Equal	E or U
FBUL	0101	Branch Unoredered or Less	L or U

Table 7.65-1 Branching conditions of FBcc instruction (2 / 2)

Mnemonic	cc	Contents	conditions (FCC field)
FBGE	1010	Branch Greater or Equal	G or E
FBL	0100	Branch Less	L
FBUGE	1011	Branch Unordered or Greater or Equal	U or G or E
FBUG	0011	Branch Unorder or Greater	G or U
FBLE	1100	Branch Less or Equal	L or E
FBG	0010	Branch Greater	G
FBULE	1101	Branch Unordered or Less or Equal	E or L or U
FBU	0001	Branch Unordered	U
FBO	1110	Branch Ordered	E or L or G

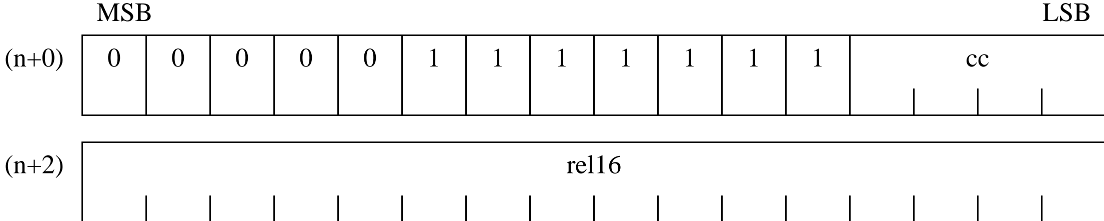
● Classification

Floating point instruction, FR81 family

● Execution Cycles

2 cycles

● Instruction Format



● EIT Occurrence and Detection

An interrupt is detected.

7.66 FBcc:D (Floating Point Conditional Branch with Delay Slot)

This is a branching instruction having a delay slot. If conditions specified for each instruction are satisfied, control branches to the address indicated by label17 relative to the program counter (PC). The rel16 value is doubled and its sign is extended. If conditions are not satisfied, no branching occurs.

● Assembler Format

FBN:D
FBA:D label17
FBNE:D label17
FBE:D label17
FBLG:D label17
FBUE:D label17
FBUL:D label17
FBGE:D label17
FBL:D label17
FBUGE:D label17
FBUG:D label17
FBLE:D label17
FBG:D label17
FBULE:D label17
FBU:D label17
FBO:D label17

The FBN:D instruction has no operand.

● Operation

if (condition) {
  $PC + 4 + exts(rel16 \times 2) \rightarrow PC$
}

The branching conditions of each instruction are shown in Table 7.66-1.

Table 7.66-1 Branching conditions of FBcc:D instruction (1 / 2)

Mnemonic	cc	Contents	conditions (FCC field)
FBN:D	0000	Branch Never	Always unsatisfied
FBA:D	1111	Branch Always	Always satisfied
FBNE:D	0111	Branch Not Equal	L or G or U
FBE:D	1000	Branch Equal	E
FBLG:D	0110	Branch Less or Greater	L or G
FBUE:D	1001	Branch Unordered or Equal	E or U
FBUL:D	0101	Branch Unoredered or Less	L or U

Table 7.66-1 Branching conditions of FBcc:D instruction (2 / 2)

Mnemonic	cc	Contents	conditions (FCC field)
FBGE:D	1010	Branch Greater or Equal	G or E
FBL:D	0100	Branch Less	L
FBUGE:D	1011	Branch Unordered or Greater or Equal	U or G or E
FBUG:D	0011	Branch Unorder or Greater	G or U
FBLE:D	1100	Branch Less or Equal	L or E
FBG:D	0010	Branch Greater	G
FBULE:D	1101	Branch Unordered or Less or Equal	E or L or U
FBU:D	0001	Branch Unordered	U
FBO:D	1110	Branch Ordered	E or L or G

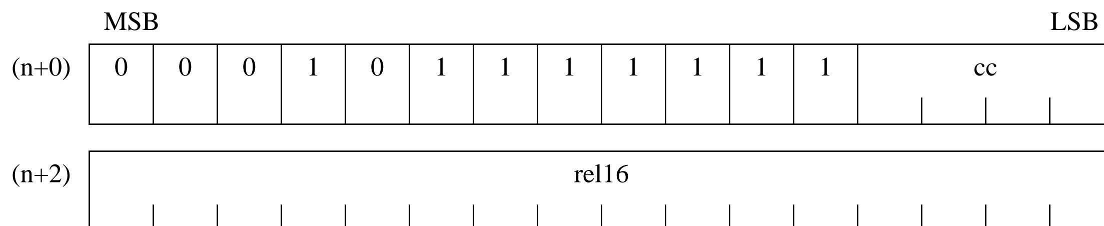
● Classification

Floating point instruction, FR81 family

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

No interrupt is detected.

7.67 FCMPs (Single Precision Floating Point Compare)

FRk and FRj are compared, and the result is reflected on the floating point condition code (FCC) of floating point control register (FCR).

- Assembler Format

FCMPs FRk, FRj

- Operation

FRk - FRj

The FCC is set as follows according to the comparison result.

FCC	Comparison result
1000_H (E)	FRk = FRj
0100_H (L)	FRk < FRj
0010_H (G)	FRk > FRj
0001_H (U)	FRk ? FRj (not possible to compare)

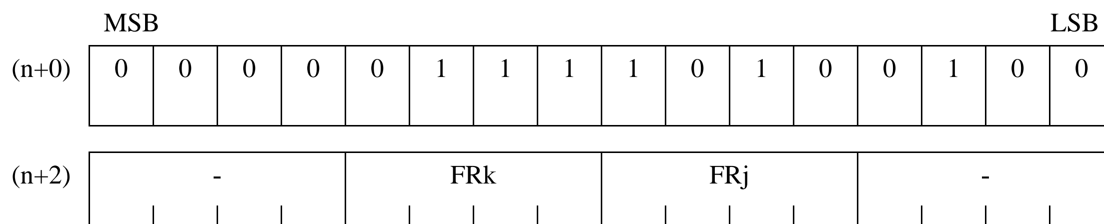
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



FR81 Family

- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- Calculation result and exception flag

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">FRj</td></tr>
<tr><td>+0</td><td>-0</td><td>+Norm</td><td>-Norm</td><td>+INF</td><td>-INF</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">FRk</td><td>+0</td><td colspan="2" rowspan="2">EQ/(-)</td><td rowspan="2">L/(-)</td><td rowspan="3">G/(-)</td><td rowspan="4">L(-)</td><td rowspan="5">G(-)</td><td rowspan="6">UO/(-)</td><td rowspan="7">UO/V</td></tr>
<tr><td>-0</td></tr>
<tr><td>+Norm</td><td colspan="2">G/(-)</td><td>*1</td></tr>
<tr><td>-Norm</td><td colspan="3">L/(-)</td><td>*1</td></tr>
<tr><td>+INF</td><td colspan="4">G/(-)</td><td>E/(-)</td></tr>
<tr><td>-INF</td><td colspan="5">L/(-)</td><td>E/(-)</td></tr>
<tr><td>QNaN</td><td colspan="7"></td></tr>
<tr><td>SNaN</td><td colspan="8"></td></tr>
</table>

*1: G/(-) or L/(-) or E/(-)

7.68 FDIVs (Single Precision Floating Point Division)

FRk is divided by FRj, and its result is stored in FRi.

- Assembler Format

FDIVs FRk, FRj, FRi

- Operation

FRk / FRj → FRi

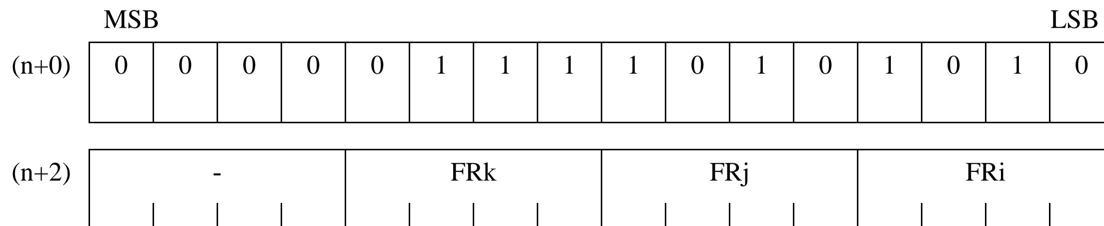
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

9 cycles

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

FR81 Family

- Calculation result and exception flag

		FRj							
		+0	-0	+Norm	-Norm	+INF	-INF	QNaN	SNaN
FRk	+0	QNaN/V		+0(-)	-0/(-)	+0/(-)	-0/(-)	QNaN/(-)	QNaN/V
	-0			-0/(-)	+0/(-)	-0/(-)	+0/(-)		
	+Norm	+INF/Z	-INF/Z	*1	*2	+0/(-)	-0/(-)		
	-Norm	-INF/Z	+INF/Z	*2	*1	-0/(-)	+0/(-)		
	+INF	+INF/(-)	-INF/(-)	+INF/(-)	-INF/(-)	QNaN/V			
	-INF	-INF/(-)	+INF/(-)	-INF/(-)	+INF/(-)				
	QNaN								
	SNaN								

*1: +INF/0, X or +0/U, X or +Norm/(X)

*2: -INF/0, X or -0/U, X or -Norm/(X)

7.69 FiTOs (Convert from Integer to Single Precision Floating Point)

A 32-bit signed integer in FRj is converted into a single-precision floating point value, and it is stored in FRi.

- Assembler Format

FiTOs FRj, FRi

- Operation

(float) FRj → FRi

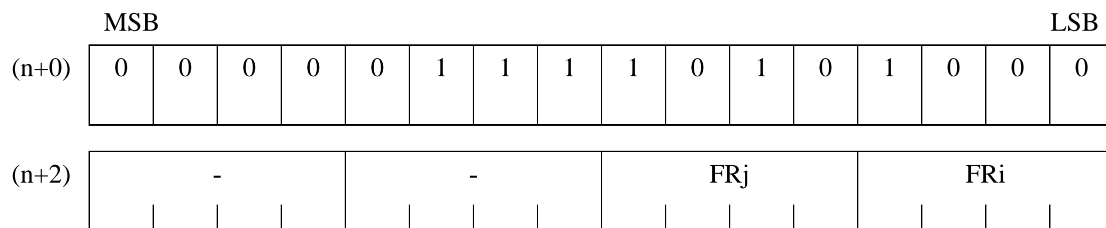
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

FR81 Family

- Calculation result and exception flag

FRj	Output result
±0	0/(-)
+Norm	+Norm/(X)
-Norm	-Norm/(X)
+MAX	+Nrom/X

7.70 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address Rj to FRi.

- Assembler Format

FLD @Rj, FRi

- Operation

(Rj) → FRi

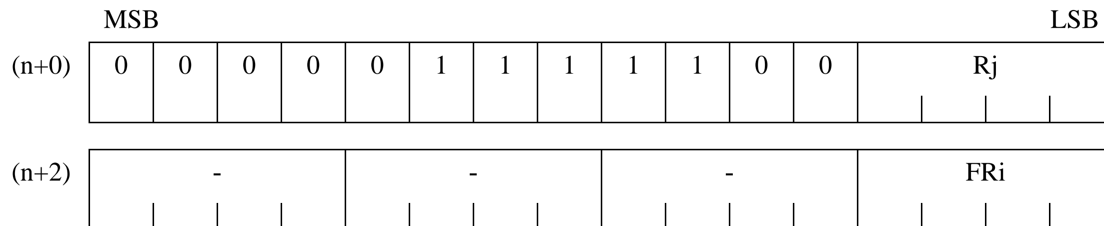
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

FR81 Family

7.71 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R13+Rj to FRi.

● Assembler Format

FLD @(R13,Rj), FRi

● Operation

(R13 + Rj) → FRi

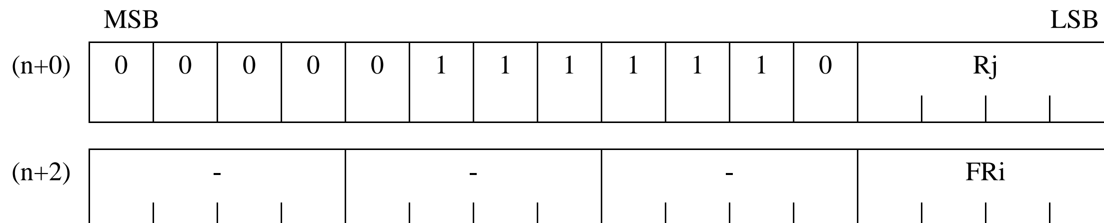
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

7.72 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R14+o14 × 4 to FRi. Signed o14 value is calculated. The value in o14 × 4 is specified as disp16.

- Assembler Format

FLD @(R14,disp16), FRi

- Operation

(R14 + o14 × 4) → FRi

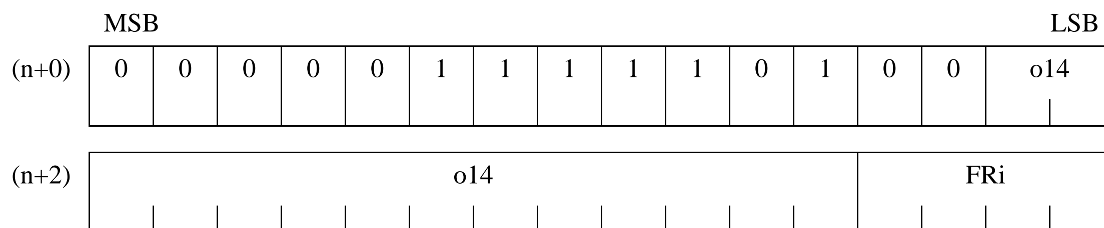
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

FR81 Family

7.73 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R15+u14x4 to FRi. Unsigned u14 value is calculated. The value in u14x4 is specified as udisp16.

● Assembler Format

FLD @(R15,udisp16), FRi

● Operation

$(R15 + u14 \times 4) \rightarrow FRi$

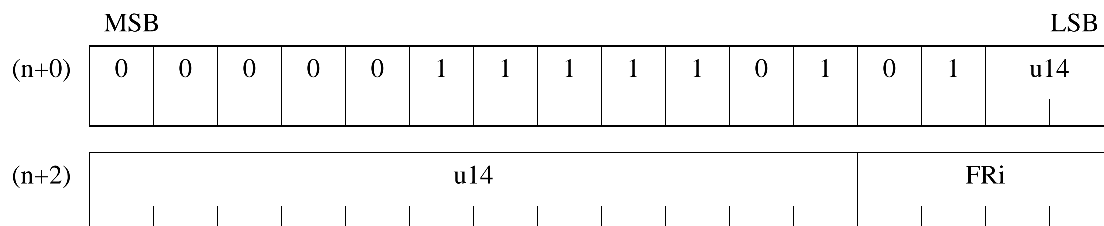
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

7.74 FLD (Single Precision Floating Point Data Load)

Loads the value at memory address R15 to FRi, and adds 4 to R15.

● Assembler Format

FLD @R15+, FRi

● Operation

(R15) → FRi

R15 + 4 → R15

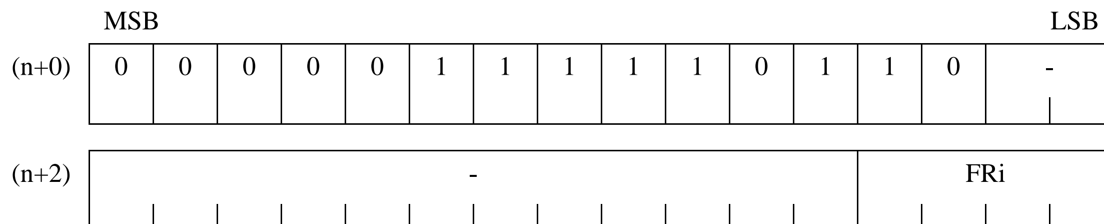
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

FR81 Family

7.75 FLD (Load Word Data in Memory to Floating Register)

Loads the word data at memory address BP+u16 × 4 to FRi. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

● Assembler Format

FLD @(BP, udisp18), FRi

● Operation

(BP+u16 × 4) → FRi

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

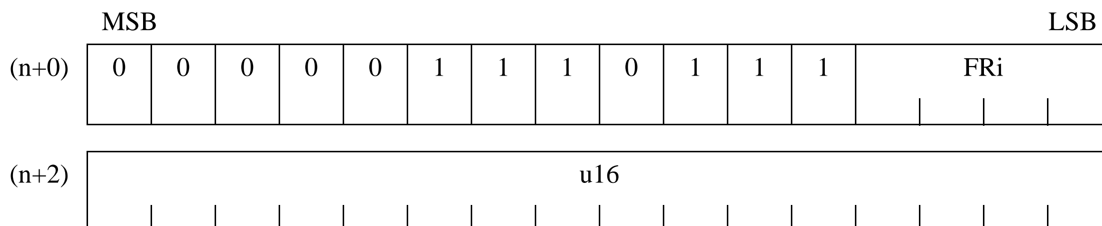
● Classification

Memory load instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

7.76 FLDM (Single Precision Floating Point Data Load to Multiple Register)

The registers shown on the frlist are sequentially restored from the stack. Registers FR0 to FR15 can be set on the frlist. They are processed in ascending order of register numbers.

● Assembler Format

FLDM (frlist)

● Operation

The following operations are repeated according to the number of registers specified in the parameter frlist.

(R15) → FRi

R15 + 4 → R15

The bit and register relation of frlist of FLDM instruction is shown in Table 7.76-1.

Table 7.76-1 The bit and register relation of frlist of FLDM instruction

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FRi	FR15	FR14	FR13	FR12	FR11	FR10	FR9	FR8	FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0

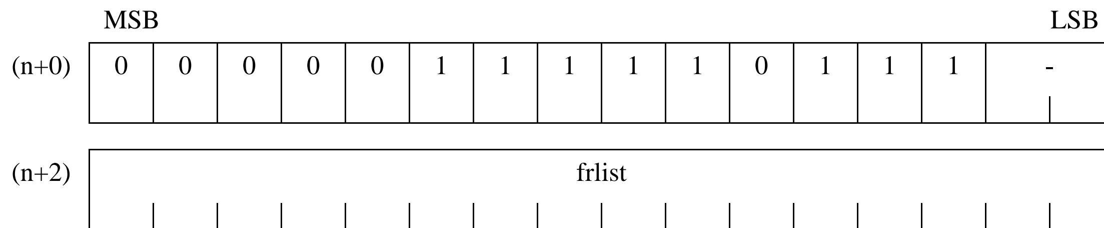
● Classification

Floating point instruction, FR81 family

● Execution Cycles

na cycle (n: Transfer register number)

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

If an exception occurs during repetition, the access that generated the exception is interrupted. The remaining values of frlist are stored in the RL of exception status register (ESR), and the exception process is executed.

7.77 FMADDs (Single Precision Floating Point Multiply and Add)

FRk is multiplied by FRj, and FRi is added to its result and then stored in FRi.

- Assembler Format

FMADDs FRk, FRj, FRi

- Operation

$FRk \times FRj + FRi \rightarrow FRi$

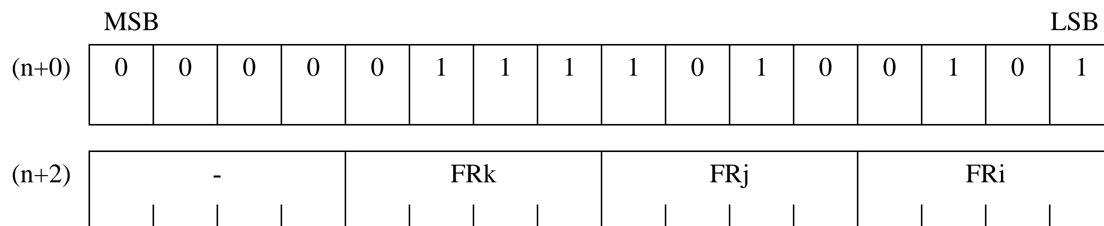
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

4 cycles

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

FR81 Family

- Calculation result and exception flag

[Multiplication]

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">FRj</td></tr>
<tr><td>+0</td><td>-0</td><td>+Norm</td><td>-Norm</td><td>+INF</td><td>-INF</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">FRk</td><td>+0</td><td>+0/(-)</td><td>-0/(-)</td><td>+0/(-)</td><td>-0/(-)</td><td colspan="2" rowspan="2">QNaN/V</td><td rowspan="6">QNaN/(-)</td><td rowspan="7">QNaN/V</td></tr>
<tr><td>-0</td><td>-0/(-)</td><td>+0/(-)</td><td>-0/(-)</td><td>+0/(-)</td></tr>
<tr><td>+Norm</td><td>+0/(-)</td><td>-0/(-)</td><td>*1</td><td>*2</td><td>+INF/(-)</td><td>-INF/(-)</td></tr>
<tr><td>-Norm</td><td>-0/(-)</td><td>+0/(-)</td><td>*2</td><td>*1</td><td>-INF/(-)</td><td>+INF/(-)</td></tr>
<tr><td>+INF</td><td colspan="2" rowspan="2">QNaN/V</td><td>+INF/(-)</td><td>-INF/(-)</td><td>+INF/(-)</td><td>-INF/(-)</td></tr>
<tr><td>-INF</td><td>-INF/(-)</td><td>+INF/(-)</td><td>-INF/(-)</td><td>+INF/(-)</td></tr>
<tr><td>QNaN</td><td colspan="7"></td></tr>
<tr><td>SNaN</td><td colspan="8"></td></tr>
</table>

*1: +INF/0, X or +0/U, X or +Norm/(X)

*2: -INF/0, X or -0/U, X or -Norm/(X)

[Addition]

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="8">FRj</td></tr>
<tr><td>+0</td><td>-0</td><td>+Norm</td><td>-Norm</td><td>+INF</td><td>-INF</td><td>QNaN</td><td>SNaN</td></tr>
<tr><td rowspan="8">FRk</td><td>+0</td><td colspan="2">+0/(-)</td><td rowspan="2">+Norm/(X)</td><td rowspan="2">-Norm/(X)</td><td rowspan="4">+INF/(-)</td><td rowspan="4">-INF/(-)</td><td rowspan="6">QNaN/(-)</td><td rowspan="7">QNaN/V</td></tr>
<tr><td>-0</td><td></td><td>-0/(-)</td></tr>
<tr><td>+Norm</td><td colspan="2">+Norm/(X)</td><td>*1</td><td>*2</td></tr>
<tr><td>-Norm</td><td colspan="2">-Norm/(X)</td><td>*2</td><td>*1</td></tr>
<tr><td>+INF</td><td colspan="5"></td><td>QNaN/V</td></tr>
<tr><td>-INF</td><td colspan="4">-INF/(-)</td><td>QNaN/V</td><td>+INF/(-)</td></tr>
<tr><td>QNaN</td><td colspan="7"></td></tr>
<tr><td>SNaN</td><td colspan="8"></td></tr>
</table>

*1: +INF/0, X or -INF/P. X or ± Norm/(X)

*2: ± 0/(-) or ± 0/U or ± Norm/(X)

7.78 FMOVs (Single Precision Floating Point Move)

Loads the value in FRj to FRi.

● Assembler Format

FMOVs FRj, FRi

● Operation

FRj → FRi

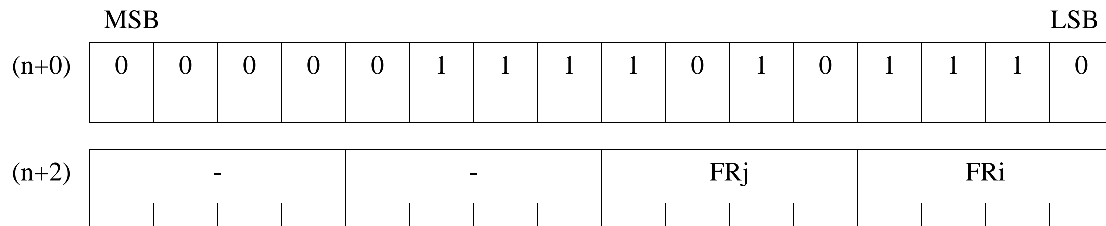
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an interrupt is detected.

FR81 Family

7.79 FMSUBs (Single Precision Floating Point Multiply and Subtract)

FRk is multiplied by FRj, and its result is subtracted by FRi and then stored in FRi.

● Assembler Format

FMSUBs FRk, FRj, FRi

● Operation

FRk × FRj - FRi → FRi

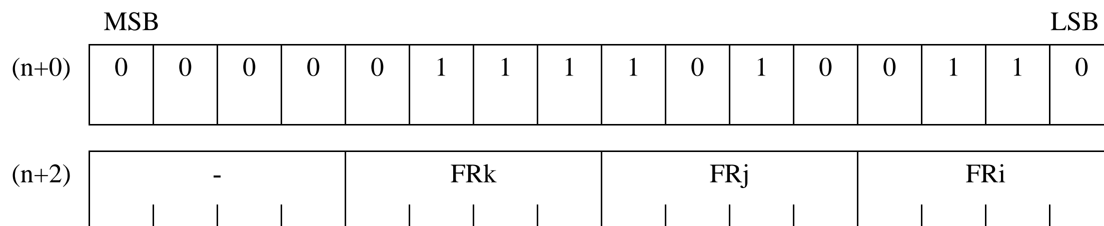
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

4 cycles

● Instruction Format



● EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- Calculation result and exception flag

[Multiplication]

		FRj							
		+0	-0	+Norm	-Norm	+INF	-INF	QNaN	SNaN
FRk	+0	+0/(-)	-0/(-)	+0/(-)	-0/(-)	QNaN/V		QNaN/(-)	QNaN/V
	-0	-0/(-)	+0/(-)	-0/(-)	+0/(-)				
	+Norm	+0/(-)	-0/(-)	*1	*2	+INF/(-)	-INF/(-)		
	-Norm	-0/(-)	+0/(-)	*2	*1	-INF/(-)	+INF/(-)		
	+INF	QNaN/V		+INF/(-)	-INF/(-)	+INF/(-)	-INF/(-)		
	-INF			-INF/(-)	+INF/(-)	-INF/(-)	+INF/(-)		
	QNaN								
	SNaN								

*1: +INF/0, X or +0/U, X or +Norm/(X)

*2: -INF/0, X or -0/U, X or -Norm/(X)

[Addition]

		FRj							
		+0	-0	+Norm	-Norm	+INF	-INF	QNaN	SNaN
FRk	+0	+0/(-)		+Norm/(X)	-Norm/(X)	+INF/(-)	-INF/(-)	QNaN/(-)	QNaN/V
	-0	-0/(-)							
	+Norm	+Norm/(X)		*1	*2				
	-Norm	-Norm/(X)		*2	*1				
	+INF	+INF/(-)				QNaN/V			
	-INF	-INF/(-)					QNaN/V		
	QNaN								
	SNaN								

*1: +INF/0 or -INF/0 or ± NorM/(X)

*2: ± 0/(-) or ± 0/U or ± Norm/(X)

FR81 Family

7.80 FMULs (Single Precision Floating Point Multiply)

FRk is multiplied by FRj, and its result is stored in FRi.

- Assembler Format

FMULs FRk, FRj, FRi

- Operation

FRk × FRj → FRi

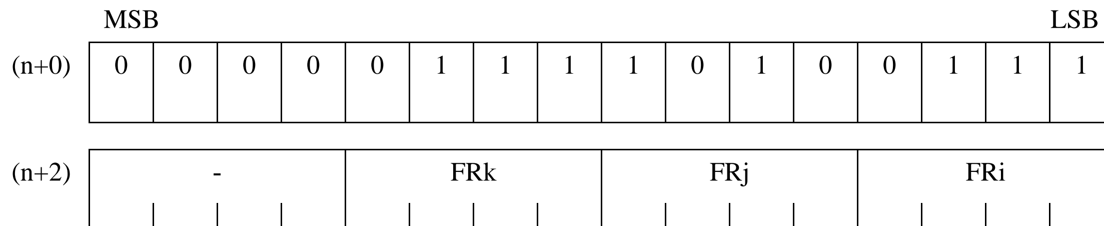
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- Calculation result and exception flag

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">FRj</th></tr>
<tr><th>+0</th><th>-0</th><th>+Norm</th><th>-Norm</th><th>+INF</th><th>-INF</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="8">FRk</td><td>+0</td><td>+0/(-)</td><td>-0/(-)</td><td>+0/(-)</td><td>-0/(-)</td><td colspan="2" rowspan="2">QNaN/V</td><td rowspan="6">QNaN/(-)</td><td rowspan="7">QNaN/V</td></tr>
<tr><td>-0</td><td>-0/(-)</td><td>+0/(-)</td><td>-0/(-)</td><td>+0/(-)</td></tr>
<tr><td>+Norm</td><td>+0/(-)</td><td>-0/(-)</td><td>*1</td><td>*2</td><td>+INF/(-)</td><td>-INF/(-)</td></tr>
<tr><td>-Norm</td><td>-0/(-)</td><td>+0/(-)</td><td>*2</td><td>*1</td><td>-INF/(-)</td><td>+INF/(-)</td></tr>
<tr><td>+INF</td><td colspan="2" rowspan="2">QNaN/V</td><td>+INF/(-)</td><td>-INF/(-)</td><td>+INF/(-)</td><td>-INF/(-)</td></tr>
<tr><td>-INF</td><td>-INF/(-)</td><td>+INF/(-)</td><td>-INF/(-)</td><td>+INF/(-)</td></tr>
<tr><td>QNaN</td><td colspan="7"></td></tr>
<tr><td>SNaN</td><td colspan="8"></td></tr>
</table>

*1: +INF/0, X or +0/U, X or +Norm/(X)

*2: -INF/0, X or 0/U, X or -Norm/(X)

FR81 Family

7.81 FNEGs (Single Precision Floating Point sign reverse)

A sign of FRj value is inverted, and the result is stored in FRi.

- Assembler Format

FNEGs FRj, FRi

- Operation

FRj × -1 → FRi

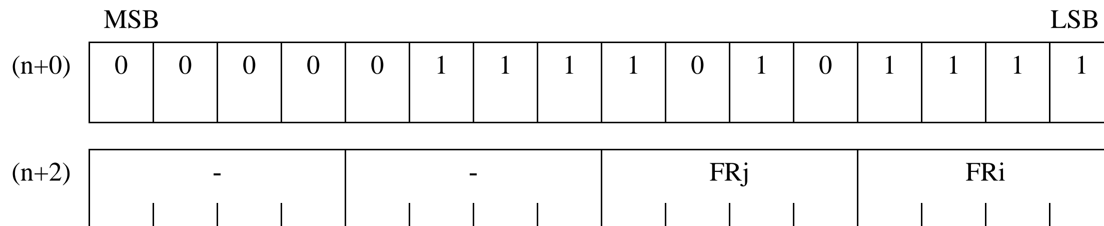
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an interrupt is detected.

7.82 FSQRTs (Single Precision Floating Point Square Root)

A square root of FRj is calculated, and its result is stored in FRi.

- Assembler Format

FSQRTs FRj, FRi

- Operation

$\sqrt{\ } \text{FRj} \rightarrow \text{FRi}$

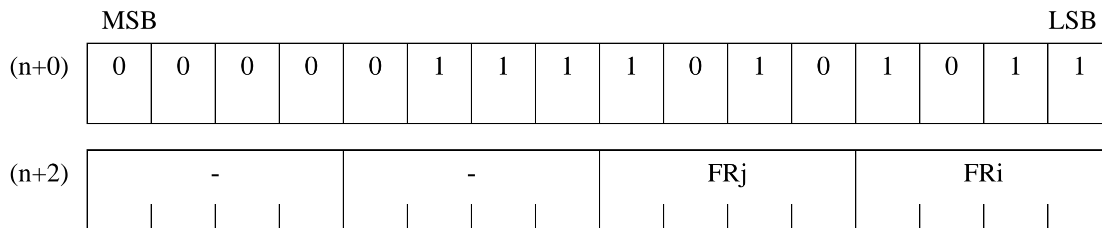
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

14 cycles

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- Calculation result and exception flag

FRj							
+0	-0	+Norm	-Norm	+INF	-INF	QNaN	SNaN
+0/(-)	-0/(-)	+Norm/(X)	QNaN/V	+INF/(-)	QNaN/V	QNaN/(-)	QNaN/V

FR81 Family

7.83 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address Rj.

- Assembler Format

FST FRi, @Rj

- Operation

FRi → (Rj)

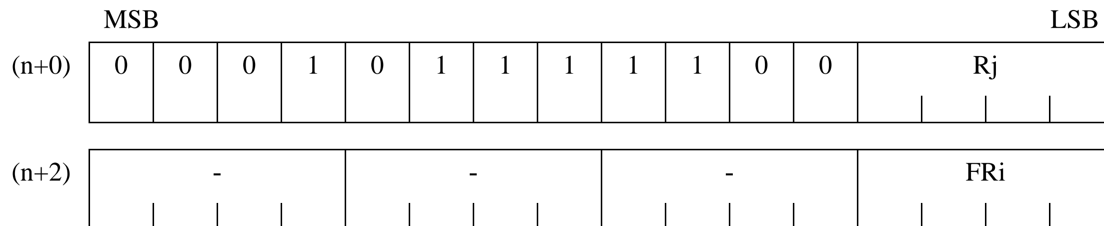
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

7.84 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address R13+Rj.

- Assembler Format

FST FRi, @(R13,Rj)

- Operation

FRi → (R13 + Rj)

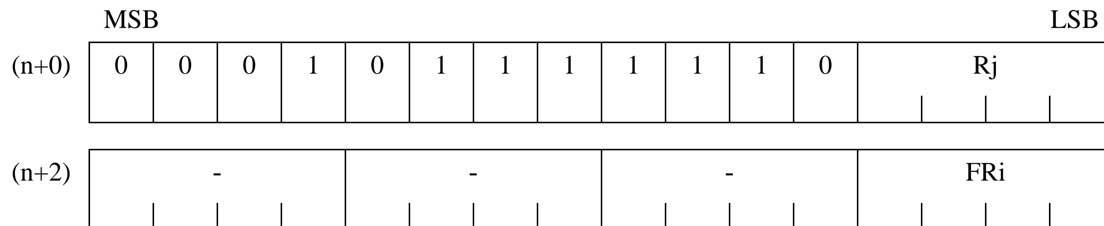
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

FR81 Family

7.85 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address R14+o14 × 4. Signed o14 value is calculated. The value in o14 × 4 is specified as disp16.

● Assembler Format

FST FRi, @(R14,disp16)

● Operation

FRi → (R14 + o14 × 4)

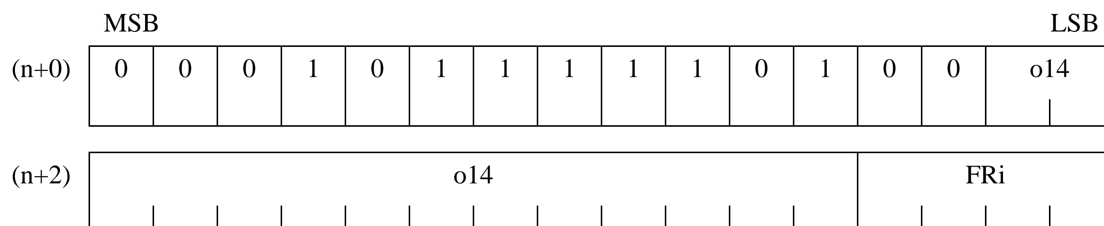
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

7.86 FST (Single Precision Floating Point Data Store)

Loads the value in FRi to memory address R15+u14 × 4. Unsigned u14 value is calculated. The value in u14 × 4 is specified as udisp16.

- Assembler Format

FST FRi, @(R15,udisp16)

- Operation

FRi → (R15 + u14 × 4)

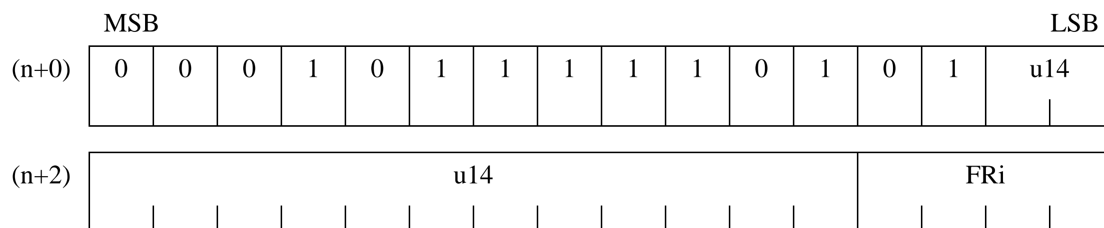
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

FR81 Family

7.87 FST (Single Precision Floating Point Data Store)

R15 is subtracted by 4, and the value in FRi is loaded to the memory address identified by new R15.

● Assembler Format

FST FRi, @-R15

● Operation

R15 - 4 → R15

FRi → (R15)

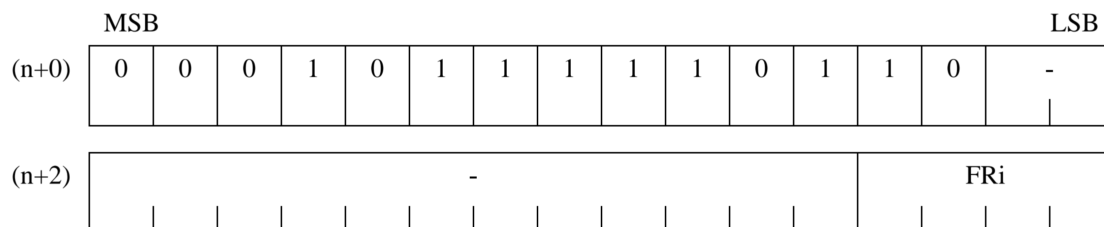
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

7.88 FST (Store Word Data in Floating Point Register to Memory)

Loads the word data in FRi to memory address BP+u16 × 4. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

● Assembler Format

FST FRi, @(BP, udisp18)

● Operation

FRi → (BP+u16 × 4)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

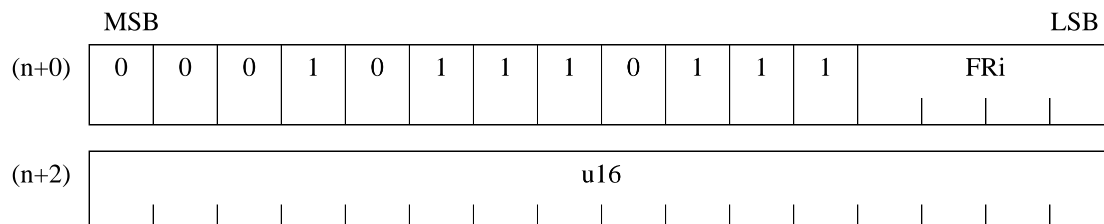
● Classification

Memory store instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

FR81 Family

7.89 FSTM (Single Precision Floating Point Data Store from Multiple Register)

The registers shown on frlist are sequentially saved in the stack. Registers FR0 to FR15 can be set on the frlist. They are processed in descending order of register numbers.

● Assembler Format

FSTM (frlist)

● Operation

The following operations are repeated according to the number of registers specified in the parameter frlist.

R15 - 4 → R15

(R15) → FRi

The bit and register relation of frlist of FSTM instruction is shown in Table 7.89-1.

Table 7.89-1 The bit and register relation of frlist of FSTM instruction

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FRi	FR0	FR1	FR2	FR3	FR4	FR5	FR6	FR7	FR8	FR9	FR10	FR11	FR12	FR13	FR14	FR15

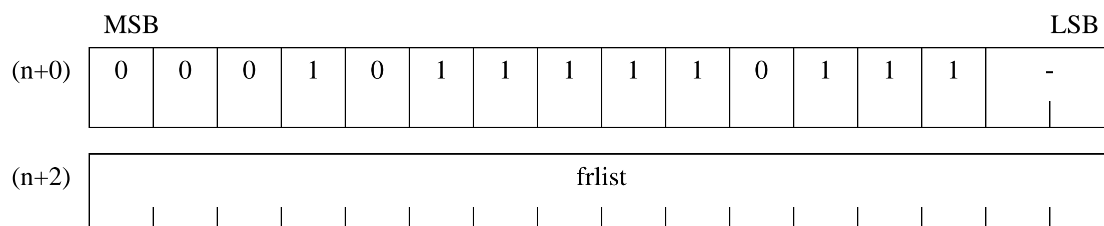
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

na cycle (n: Transfer register number)

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error or FPU absence error), or an interrupt is detected.

If an exception occurs during repetition, the access that generated the exception is interrupted. The remaining values of frlist are stored in the register list (RL) of exception status register (ESR), and the exception process is executed.

FR81 Family

7.90 FsTOi (Convert from Single Precision Floating Point to Integer)

A single-precision floating point value of FRj is converted into a 32-bit signed integer, and it is stored in FRi.

- Assembler Format

FsTOi FRj, FRi

- Operation

(int) FRj → FRi

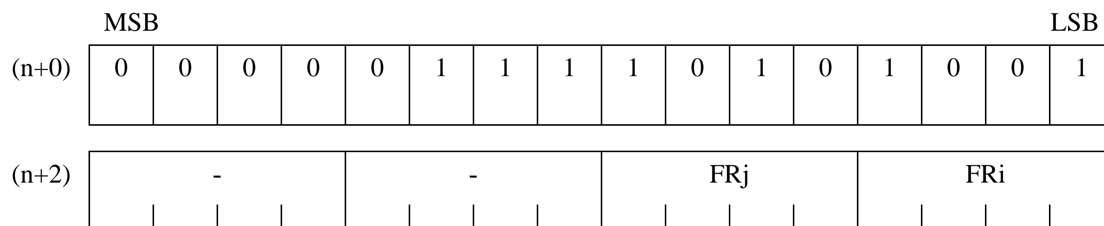
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- Calculation result and exception flag

FRj	Output result
± 0	0/(-)
± Den	0/D
+Norm	+0/(X), +Norm/(X), +MAX/V
-Norm	+0/(X), -Norm/(X), -MAX/V
+INF	+MAX/V
-INF	-MAX/V
QNaN, SNaN	± MAX/V

7.91 FSUBs (Single Precision Floating Point Subtract)

FRk is subtracted by FRj, and its result is stored in FRi.

● Assembler Format

FSUBs FRk, FRj, FRi

● Operation

FRk - FRj → FRi

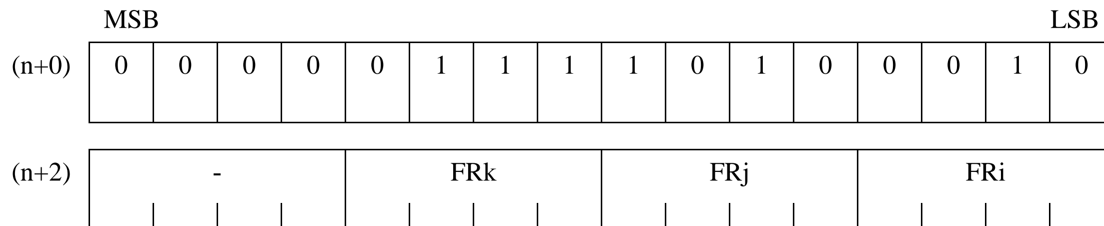
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

An invalid instruction exception (FPU absence error), an FPU exception, or an interrupt is detected.

- Calculation result and exception flag

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="8">FRj</th></tr>
<tr><th>+0</th><th>-0</th><th>+Norm</th><th>-Norm</th><th>+INF</th><th>-INF</th><th>QNaN</th><th>SNaN</th></tr>
<tr><td rowspan="8">FRk</td><td>+0</td><td colspan="2">+0/(-)</td><td rowspan="2">+Norm/(X)</td><td rowspan="2">-Norm/(X)</td><td rowspan="4">+INF/(-)</td><td rowspan="5">-INF/(-)</td><td rowspan="6">QNaN/(-)</td><td rowspan="7">QNaN/V</td></tr>
<tr><td>-0</td><td>-0/(-)</td><td></td></tr>
<tr><td>+Norm</td><td colspan="2">+Norm/(X)</td><td>*1</td><td>*2</td></tr>
<tr><td>-Norm</td><td colspan="2">-Norm/(X)</td><td>*2</td><td>*1</td></tr>
<tr><td>+INF</td><td colspan="4">+INF/(-)</td><td>QNaN/V</td></tr>
<tr><td>-INF</td><td colspan="5">-INF/(-)</td><td>QNaN/V</td></tr>
<tr><td>QNaN</td><td colspan="7"></td></tr>
<tr><td>SNaN</td><td colspan="8"></td></tr>
</table>

*1: +INF/0 or -INF/0 or ± NorM/(X)

*2: ± 0/(-) or ± 0/U or ± Norm/(X)

FR81 Family

7.92 INT (Software Interrupt)

This is a software interrupt instruction. Reads the vector table for the interrupt vector number u8 to determine the branch destination address, and branches.

● Assembler Format

INT #u8

Vector numbers 9 to 13, 64 and 65 are used by emulators for debugging interrupts and therefore the corresponding numbers "INT #9" to "INT #13", "INT #64", "INT #65" should not be used in user programs.

● Operation

SSP-4 → SSP

PS → (SSP)

SSP-4 → SSP

PC+2 → (SSP)

"0" → CCR:I

"0" → CCR:S

(TBR+$3FC_H$-u8 × 4) → PC

Stores the values of the program counter (PC) and program status (PS) to the stack indicated by the system stack pointer (SSP) for interrupt processing. Writes "0" to the S flag in the condition code register (CCR), and uses the SSP as the stack pointer for following steps. Writes "0" to the I flag (interrupt enable flag) in the CCR to disable external interrupts. Reads the vector table for the interrupt vector number u8 to determine the branch destination address, and branches.

● Flag Change

S	I	N	Z	V	C
0	0	-	-	-	-

N, Z, V, C: Unchanged.

S, I: Cleared.

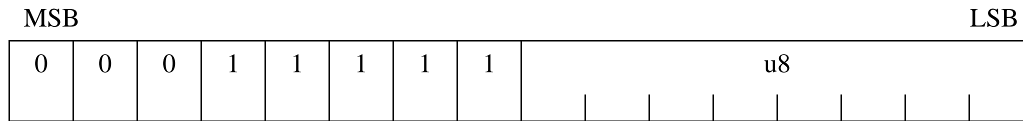
● Classification

Non-Delayed Branching instruction

● Execution Cycles

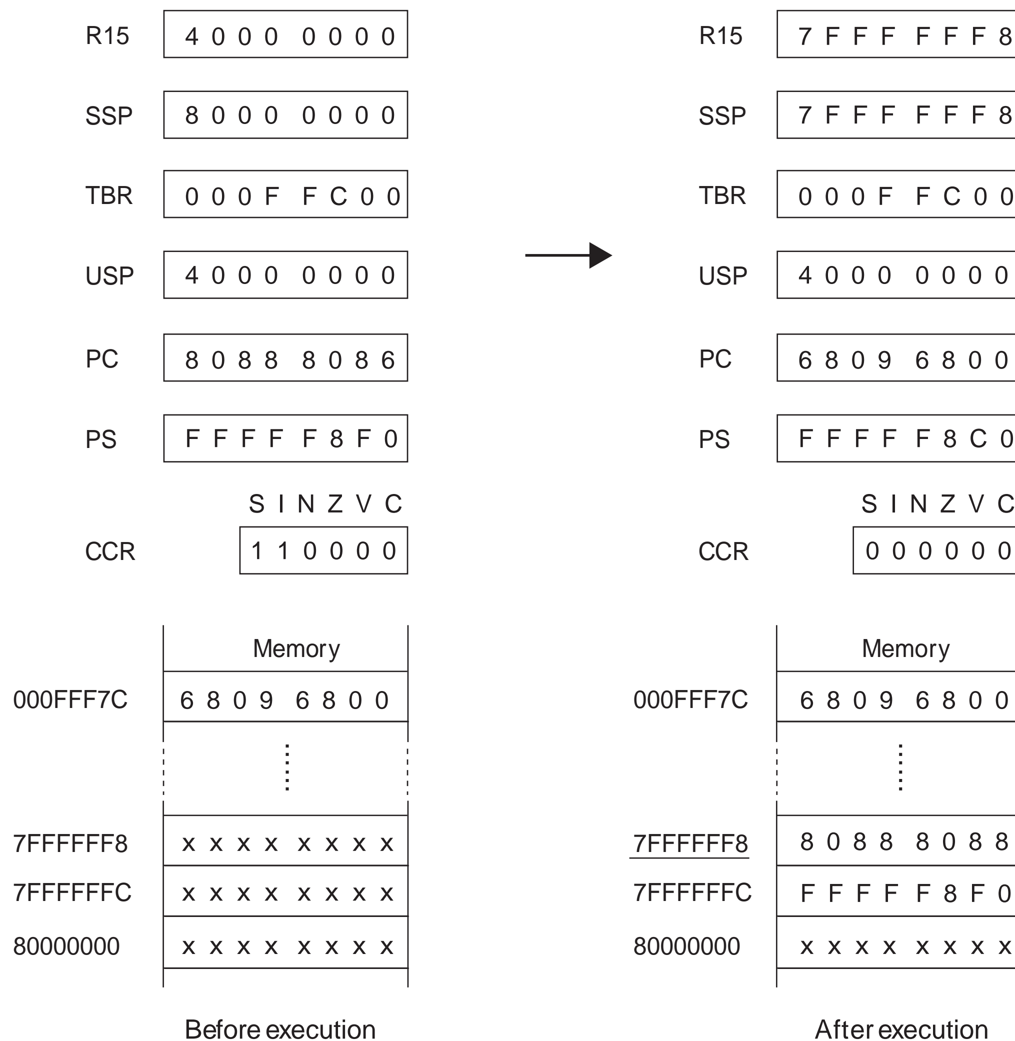
1+3a cycles

● Instruction Format



● Execution Example

INT #20H ; Bit pattern of the instruction: 0001 1111 0010 0000



7.93 INTE (Software Interrupt for Emulator)

This software interrupt instruction is used for debugging. It determines the branch destination address by reading interrupt vector number "#9" from the vector table, then branches.

● Assembler Format

INTE

● Operation

SSP-4 → SSP

PS → (SSP)

SSP-4 → SSP

PC+2 → (SSP)

4 → ILM

"0" → CCR:S

(TBR+3D8$_H$) → PC

It stores the values of the program counter (PC) and program status (PS) to the stack indicated by the system stack pointer (SSP) for interrupt processing. It writes "0" to the S flag in the condition code register (CCR), and uses the SSP as the stack pointer for the following steps. It determines the branch destination address by reading interrupt vector number #9 from the vector table, then branches.

There is not change to the I flag in the condition code register (CCR). The interrupt level mask register (ILM) in the program status (PS) is set to level 4.

● Flag Change

S	I	N	Z	V	C
0	-	-	-	-	-

I, N, Z, V, C: Unchanged.

S: Cleared.

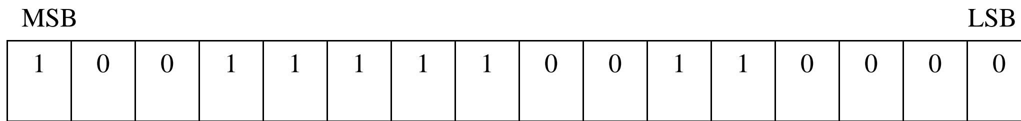
● Classification

Non-Delayed Branching instruction

● Execution Cycles

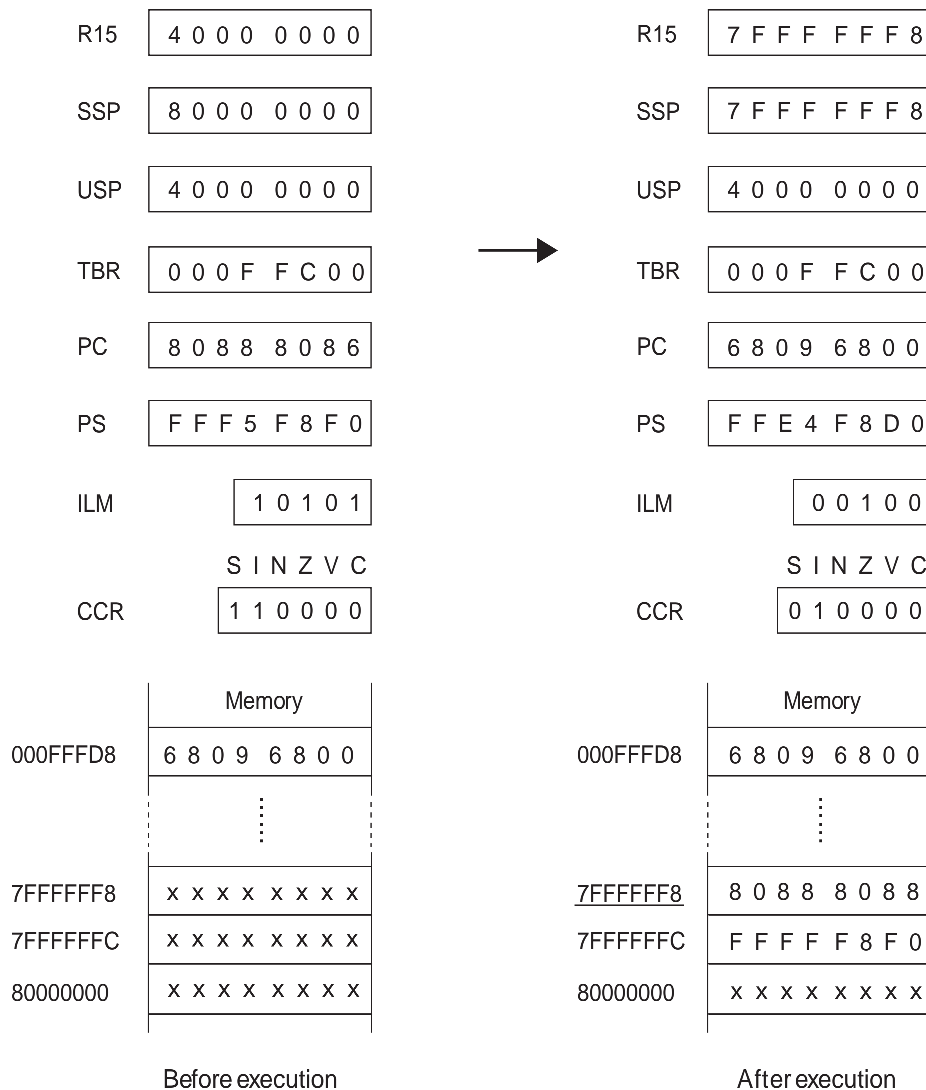
1+3a cycles

● Instruction Format



● Execution Example

INTE ; Bit pattern of the instruction: 1001 1111 0011 0000



FR81 Family

7.94 JMP (Jump)

This is a branching instruction without a delay slot. Branches to the address indicated in Ri.

- Assembler Format

JMP @Ri

- Operation

Ri → PC

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

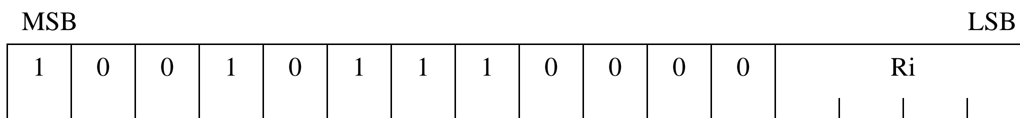
- Classification

Non-Delayed Branching instruction

- Execution Cycles

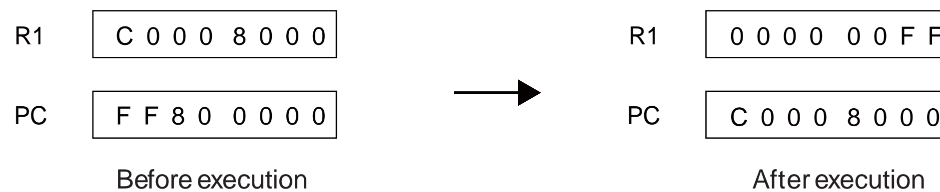
2 cycles

- Instruction Format



● Execution Example

JMP @R1 ; Bit pattern of the instruction: 1001 0111 0000 0001



FR81 Family

7.95 JMP:D (Jump)

This is a branching instruction with delay slot. Branches to the address indicated by Ri.

- Assembler Format

JMP:D @Ri

- Operation

Ri → PC

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Delayed Branching instruction

- Execution Cycles

1 cycle

- Instruction Format

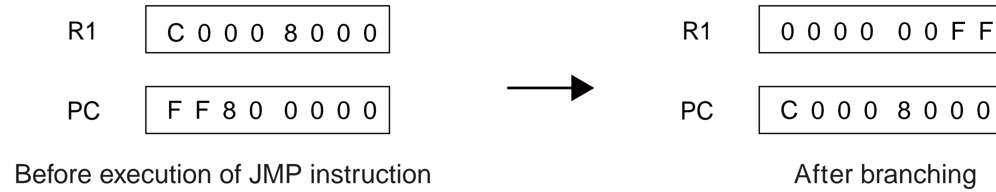
MSB LSB

1	0	0	1	1	1	1	1	0	0	0	0	Ri			

● Execution Example

JMP:D @R1 ; Bit pattern of the instruction: 1001 1111 0000 0001

LDI:8 #0FFH, R1 ; Instruction placed in delay slot



Before execution of JMP instruction After branching

The instruction placed in the delay slot will be executed before execution of the branch destination instruction. The value R1 above will vary according to the specifications of the LDI:8 instruction placed in the delay slot.

FR81 Family

7.96 LCALL (Long Call Subroutine)

This is a branching instruction without a delay slot. The next instruction address is stored in the return pointer (RP), and then control branches to the address identified by label21 relative to the program counter (PC). The value in rel20 is doubled during address calculation, and its sign is extended.

● Assembler Format

LCALL label21

● Operation

PC + 4 $\rightarrow$ RP

PC + 4 + exts(rel20 $\times$ 2) $\rightarrow$ PC

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

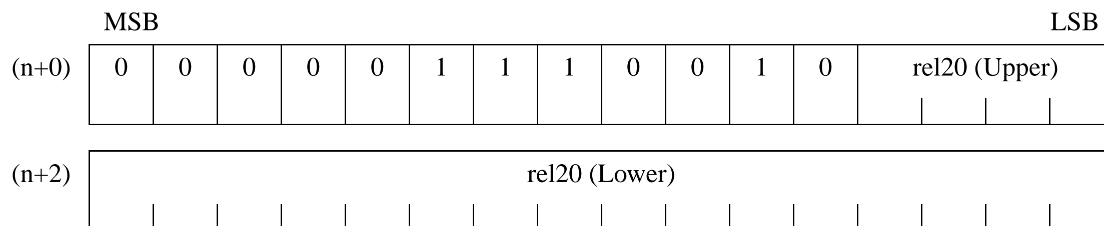
● Classification

Branching instruction without delay, FR81 family

● Execution Cycles

2 cycles

● Instruction Format



● EIT Occurrence and Detection

An interrupt is detected.

7.97 LCALL:D (Long Call Subroutine)

This is a branching instruction with a delay slot. The next instruction address is stored in the return pointer (RP), and then control branches to the address identified by label21 relative to the program counter (PC). The value in rel20 is doubled during address calculation, and its sign is extended.

● Assembler Format

LCALL:D label21

● Operation

PC + 6 → RP

PC + 4 + exts(rel20 × 2) → PC

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

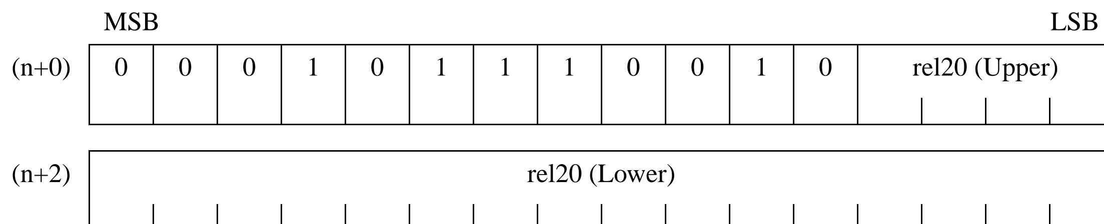
● Classification

Branching instruction with delay, FR81 family

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

No interrupt is detected.

FR81 Family

7.98 LD (Load Word Data in Memory to Register)

Loads the word data at memory address Rj to Ri.

- Assembler Format

LD @Rj, Ri

- Operation

(Rj) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

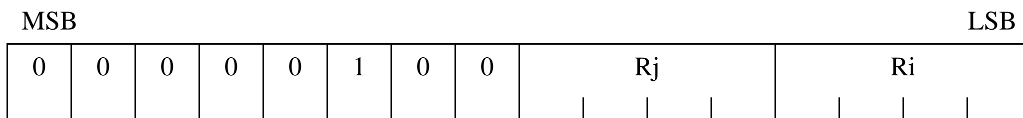
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

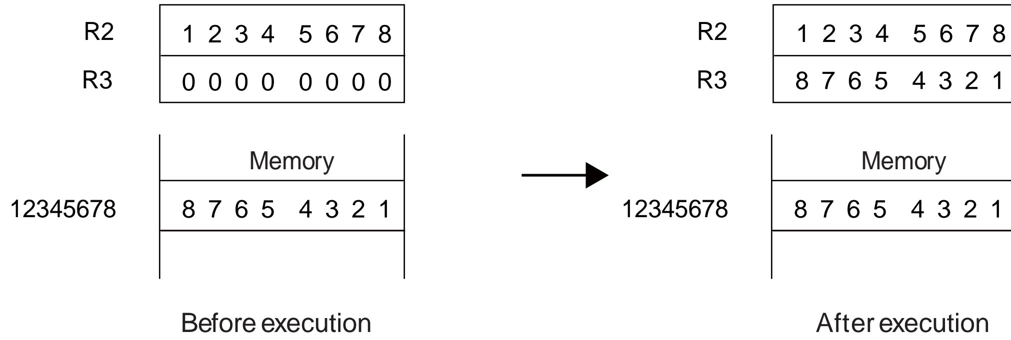
b cycle

- Instruction Format



● Execution Example

```
LD   @R2, R3          ; Bit pattern of the instruction: 0000 0100 0010 0011
```



FR81 Family

7.99 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R13 + Rj to Ri.

● Assembler Format

LD @(R13, Rj), Ri

● Operation

(R13+Rj) → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

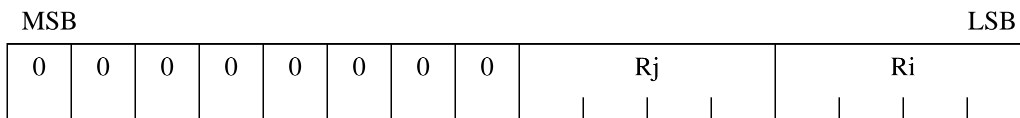
● Classification

Memory Load instruction, Instruction with delay slot

● Execution Cycles

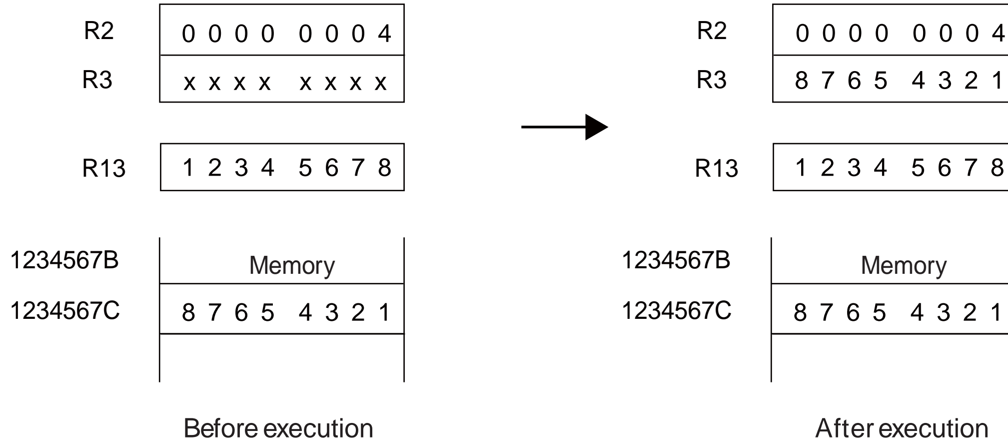
b cycle

● Instruction Format



● Execution Example

LD @(R13, R2), R3 ; Bit pattern of the instruction: 0000 0000 0010 0011



FR81 Family

7.100 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R14 + o8 × 4 to Ri. The value of o8 × 4 is specified as disp10.

- Assembler Format

LD @(R14, disp10), Ri

- Operation

(R14+o8 × 4) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

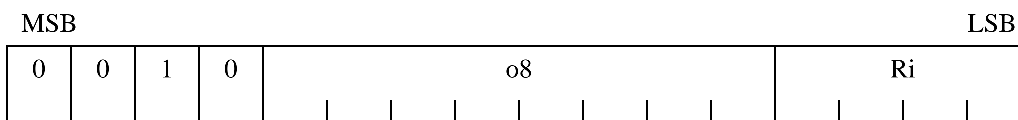
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

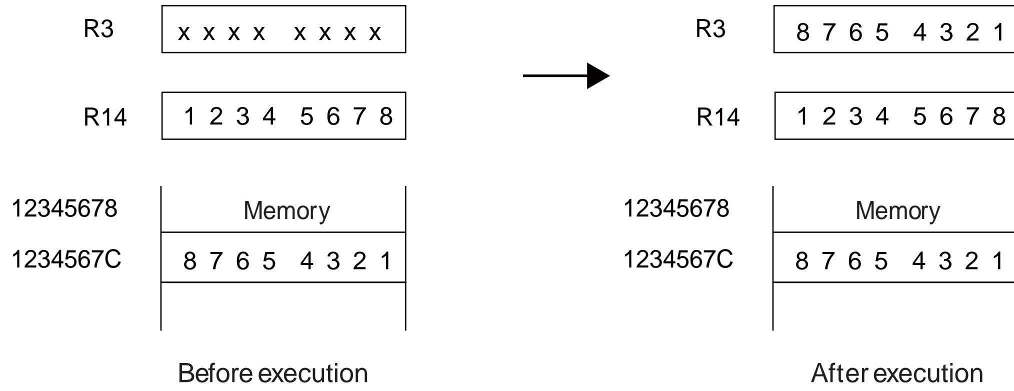
b cycle

- Instruction Format



- Execution Example

LD @(R14,4), R3 ; Bit pattern of the instruction: 0010 0000 0001 0011



FR81 Family

7.101 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R15 + u4 × 4 to Ri. The value u4 is an unsigned calculation. The value of u4 × 4 is specified as udisp6.

- Assembler Format

LD @(R15, udisp6), Ri

- Operation

(R15+u4 ×4) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

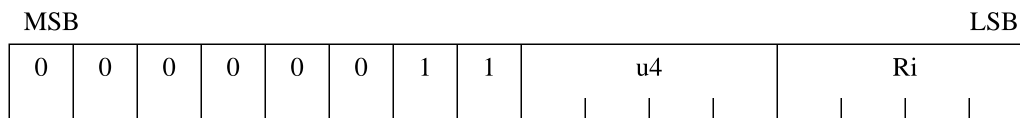
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

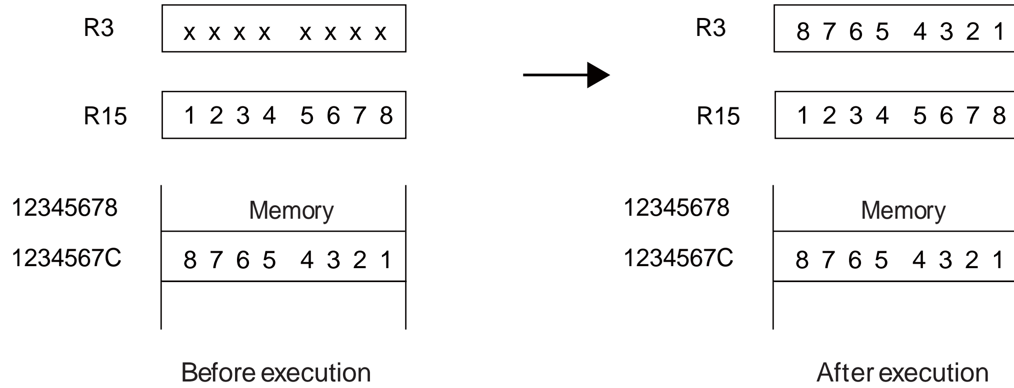
b cycle

- Instruction Format



- Execution Example

```
LD    @(R15,4), R3        ; Bit pattern of the instruction: 0000 0011 0001 0011
```



FR81 Family

7.102 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R15 to Rj, and adds 4 to the value of R15. If R15 is given as parameter Ri, the value read from the memory will be loaded into memory address R15.

● Assembler Format

LD @R15+, Ri

● Operation

(R15) → Ri

R15 + 4 → R15

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

Memory Load instruction, Instruction with delay slot

● Execution Cycles

b cycle

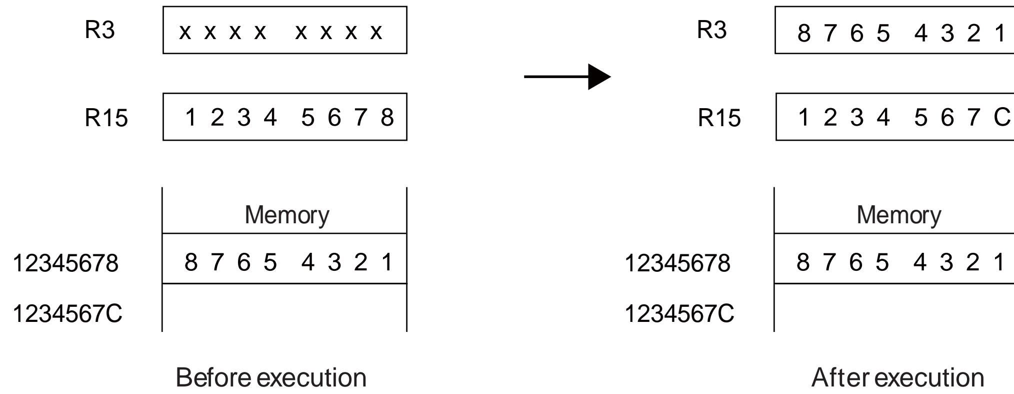
● Instruction Format

MSB ... LSB

0	0	0	0	0	1	1	1	0	0	0	0	Ri

● Execution Example

LD @R15+, R3 ; Bit pattern of the instruction: 0000 0111 0000 0011



FR81 Family

7.103 LD (Load Word Data in Memory to Register)

Loads the word data at memory address BP+u16 × 4 to Ri. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

- Assembler Format

LD @(BP, udisp18), Ri

- Operation

(BP+u16 × 4) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

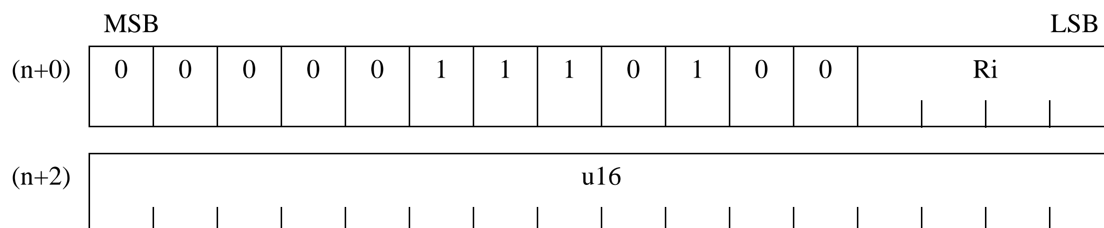
- Classification

Memory load instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (a data access error), or an interrupt is detected.

7.104 LD (Load Word Data in Memory to Register)

Loads the word data at memory address R15 to dedicated register Rs, and adds 4 to the value of R15.

- Assembler Format

LD @R15+, Rs

- Operation

(R15) → Rs

R15 + 4 → R15

If TBR, SSP, or ESR is specified in user mode or if a non-existing register number is specified in user mode, an invalid instruction exception (system-only register access) is generated.

There is no restriction in privilege mode. If a number without a dedicated register is specified for Rs in privilege mode, a value being read from memory is abandoned.

If Rs is designated as the system stack pointer (SSP) or user stack pointer (USP), and that pointer is indicating R15 (the S flag in the condition code register (CCR) is set to "0" to indicate the SSP, and to "1" to indicate the USP), the last value remaining in R15 will be the value read from memory.

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

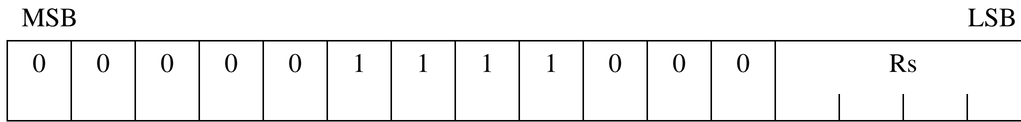
Memory Load instruction, Instruction with delay slot, FR81 updating

- Execution Cycles

b cycle

FR81 Family

● Instruction Format



● EIT Occurrence and Detection

User mode:

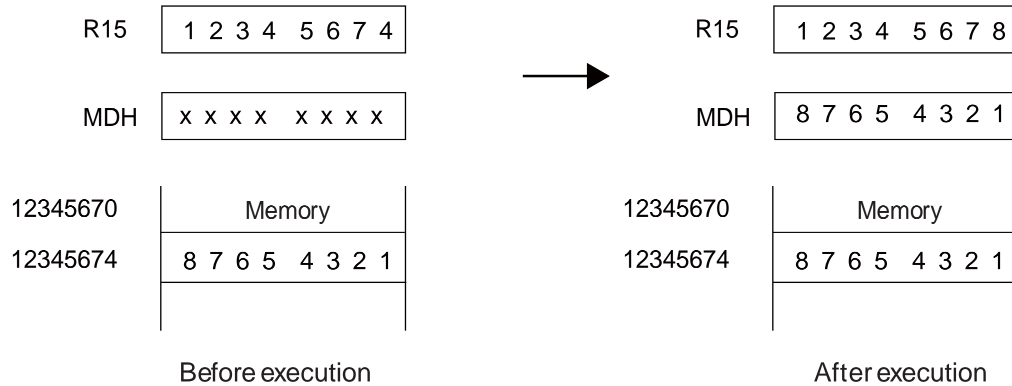
An invalid instruction exception (system-only register access) is generated. A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

Privilege mode:

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

● Execution Example

LD @R15+, MDH ; Bit pattern of the instruction: 0000 0111 1000 0100



7.105 LD (Load Word Data in Memory to Program Status Register)

Loads the word data at memory address R15 to the program status (PS), and adds 4 to the value of R15.

● Assembler Format

LD @R15+, PS

● Operation

(R15) → PS

R15 + 4 → R15

The contents of system status register (SSR) cannot be changed by this instruction regardless of the selected operation mode. If this instruction is executed in user mode, only the D1, D0, N, Z, V and C flags can be changed. The other flag values are not updated. Any bit other than SSR can be changed in privilege mode.

At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded from memory, the value 16 will be added to that data before being transferred to the ILM. IF the original ILM value is in the range 0 to 15, then any value from 0 to 31 can be transferred to the ILM.

● Flag Change

N	Z	V	C
C	C	C	C

N, Z, V, C: Data of (R15) is transferred.

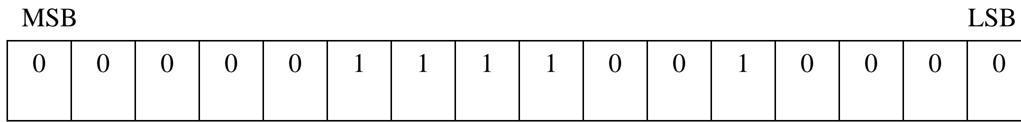
● Classification

Memory Load instruction, FR81 updating

● Execution Cycles

1+a cycles

● Instruction Format

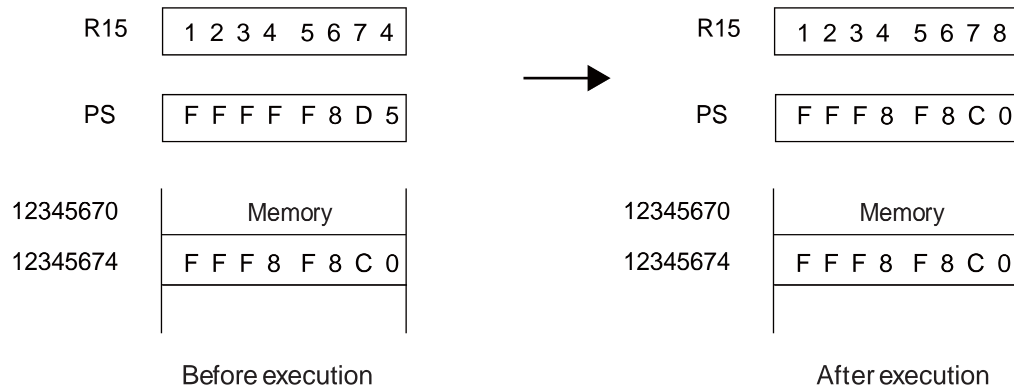


● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected. If the interrupt level mask register (ILM) or interrupt enable flag (I) is changed, an interrupt is detected using the changed values.

● Execution Example

LD @R15+, PS ; Bit pattern of the instruction: 0000 0111 1001 0000



7.106 LDI:20 (Load Immediate 20bit Data to Destination Register)

Extends the 20-bit immediate data with 12 zeros in the higher bits, loads to Ri.

- Assembler Format

LDI:20 #i20, Ri

- Operation

extu(i20) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

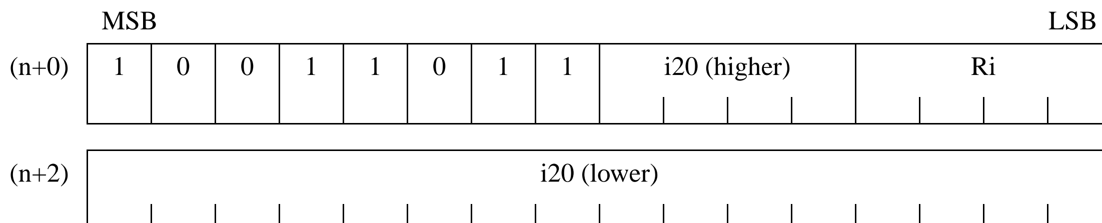
- Classification

Immediate Data Transfer instruction

- Execution Cycles

2 cycles

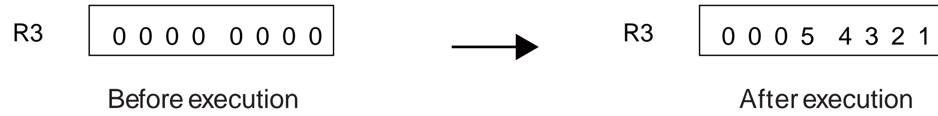
- Instruction Format



FR81 Family

- Execution Example

```
LDI:20   #54321H, R3      ; Bit pattern of the instruction:  1001 1011 0101 0011
                          ;                                  0100 0011 0010 0001
```



7.107 LDI:32 (Load Immediate 32 bit Data to Destination Register)

Loads 1 word of immediate data to Ri.

- Assembler Format

LDI:32 #i32, Ri

- Operation

i32 → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

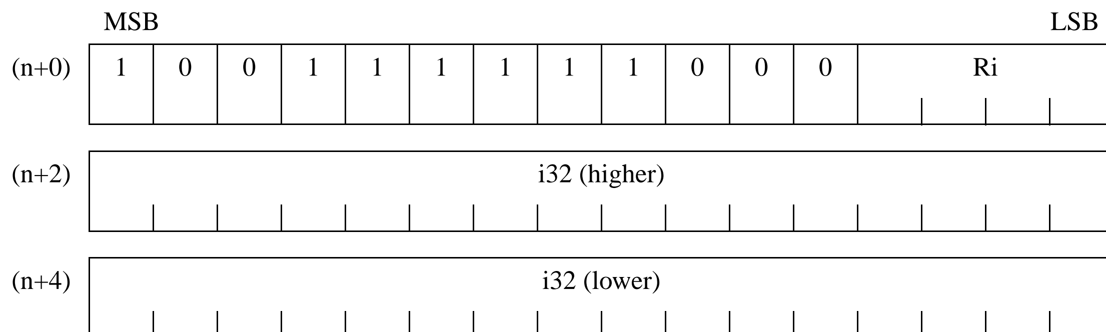
- Classification

Immediate Data Transfer instruction

- Execution Cycles

d cycle

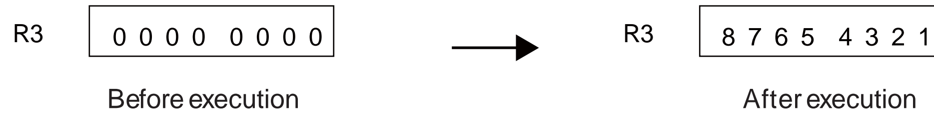
- Instruction Format



FR81 Family

- Execution Example

```
LDI:32   #87654321H, R3   ; Bit pattern of the instruction:  1001 1111 1000 0011
                          ;                                  1000 0111 0110 0101
                          ;                                  0100 0011 0010 0001
```



7.108 LDI:8 (Load Immediate 8bit Data to Destination Register)

Extends the 8-bit immediate data with 24 zeros in the higher bits, loads to Ri.

● Assembler Format

LDI:8 #i8, Ri

● Operation

extu(i8) → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

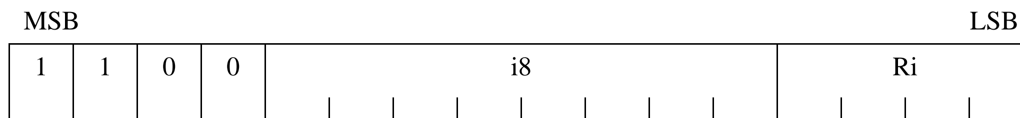
● Classification

Immediate Data Transfer instruction, Instruction with delay slot

● Execution Cycles

1 cycle

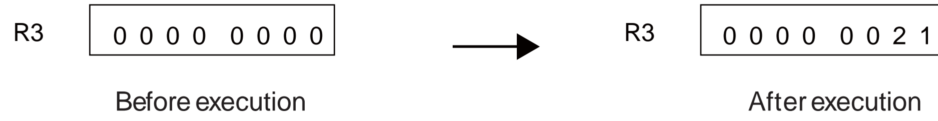
● Instruction Format



FR81 Family

- Execution Example

LDI:8 #21H, R3 ; Bit pattern of the instruction: 1100 0010 0001 0011



7.109 LDM0 (Load Multiple Registers)

The LDM0 instruction stores the word data from the address R15 to the registers in the range R0 to R7 as members of the parameter reglist and repeats the operation of adding 4 to R15. Registers are processed in ascending numerical order.

● Assembler Format

LDM0 (reglist)

Registers from R0 to R7 are separated in reglist, multiple register are arranged and specified.

● Operation

The following operations are repeated according to the number of registers specified in the parameter reglist.

(R15) → Ri

R15+4 → R15

Bit values and register numbers for reglist (LDM0) are shown in Table 7.109-1.

Table 7.109-1 Bit values and register numbers for reglist (LDM0)

Bit	Register
7	R7
6	R6
5	R5
4	R4
3	R3
2	R2
1	R1
0	R0

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Other instructions

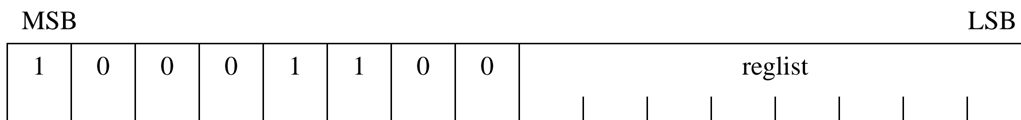
- Execution Cycles

If "n" is the number of registers specified in the parameter reglist, the execution cycles required are as follows.

When n=0: 1 cycle

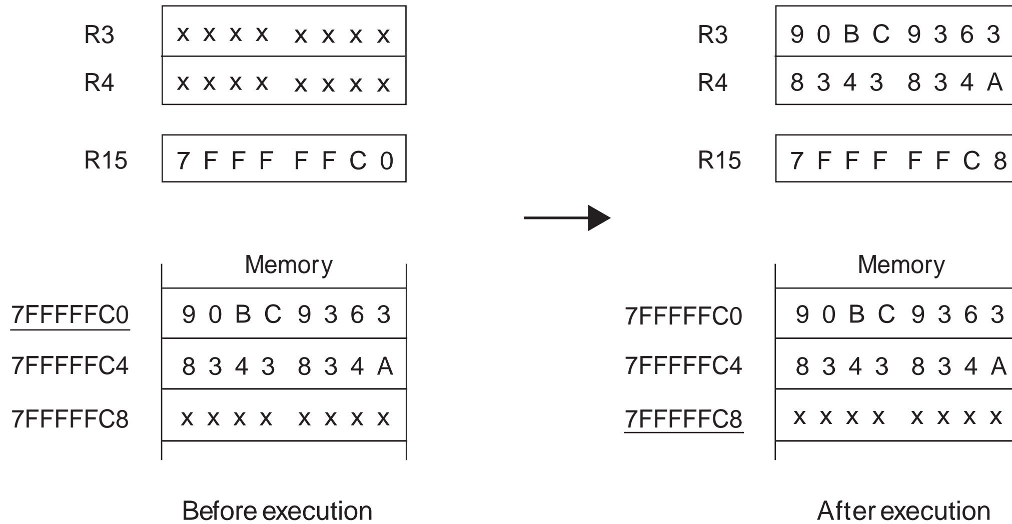
Otherwise: b × n cycles

- Instruction Format



- Execution Example

LDM0 (R3, R4) ; Bit pattern of the instruction: 1000 1100 0001 1000



7.110 LDM1 (Load Multiple Registers)

Loads the word data of address R15 to multiple registers R8 to R15 specified in reglist, repeats the operation of adding 4 to R15. Registers are processed in ascending numerical order. If R15 is specified in the parameter reglist, the final contents of R15 will be read from memory.

● Assembler Format

LDM1 (reglist)

Registers from R8 to R15 are separated in reglist, multiple register are arranged and specified.

● Operation

The following operations are repeated according to the number of registers specified in the parameter reglist.

(R15) → Ri

R15+4 → R15

Bit values and register numbers for reglist (LDM1) are shown in Table 7.110-1.

Table 7.110-1 Bit values and register numbers for reglist (LDM1)

Bit	Register
7	R15
6	R14
5	R13
4	R12
3	R11
2	R10
1	R9
0	R8

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

FR81 Family

Classification

Other instructions

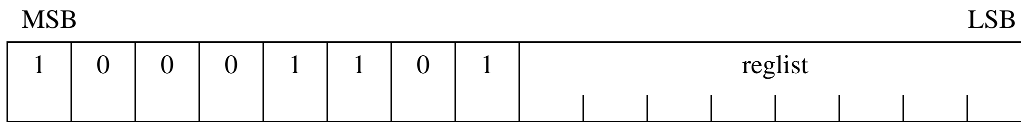
Execution Cycles

If "n" is the number of registers specified in the parameter reglist the execution cycles required are as follows.

When n=0: 1 cycle

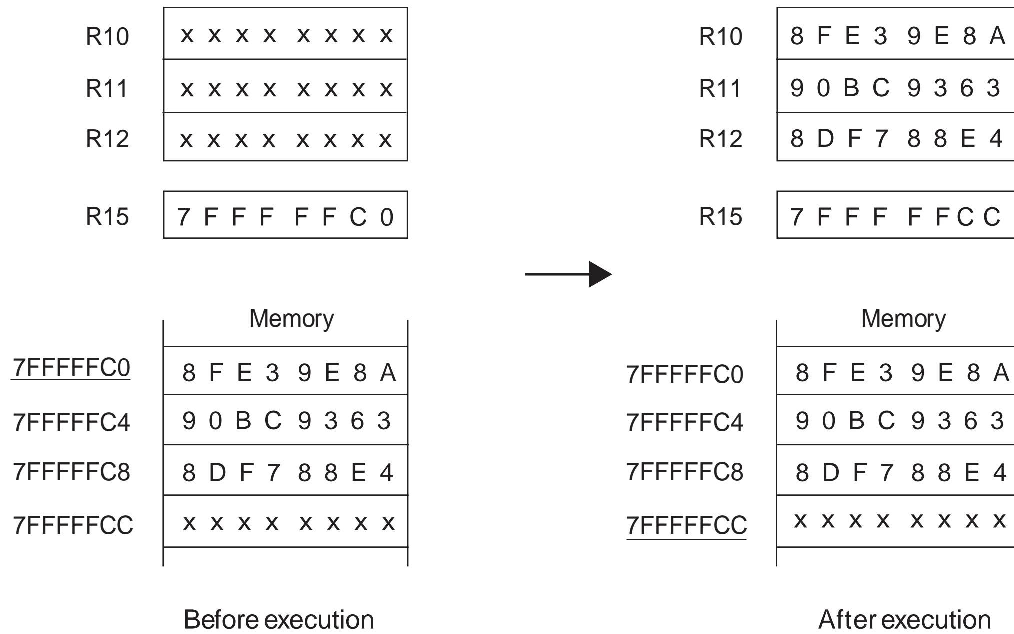
Otherwise: b × n cycles

Instruction Format



Execution Example

LDM1 (R10, R11, R12) ; Bit pattern of the instruction: 1000 1101 0001 1100



7.111 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address Rj, loads to Ri.

- Assembler Format

LDUB @Rj, Ri

- Operation

extu((Rj)) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

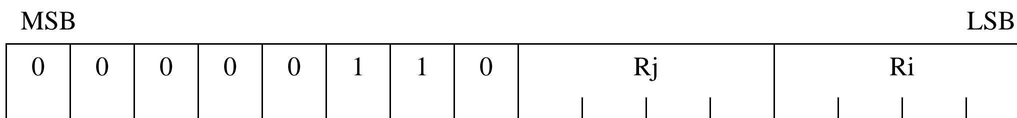
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

b cycle

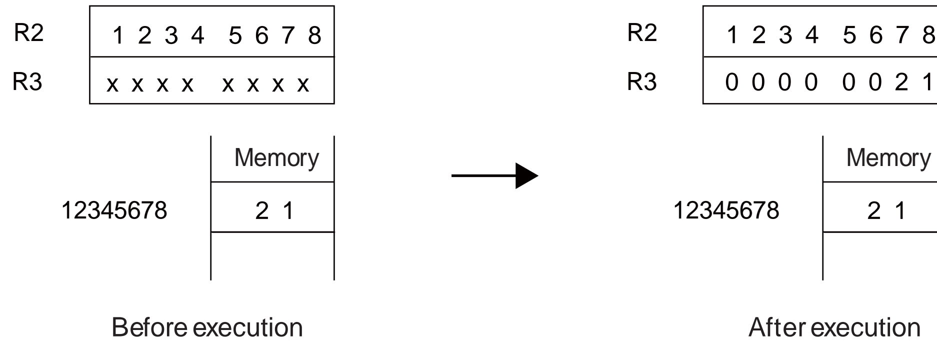
- Instruction Format



FR81 Family

- Execution Example

LDUB @R2, R3 ; Bit pattern of the instruction: 0000 0110 0010 0011



7.112 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address R13 + Rj, loads to Ri.

- Assembler Format

LDUB @(R13, Rj), Ri

- Operation

extu((R13+Rj)) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

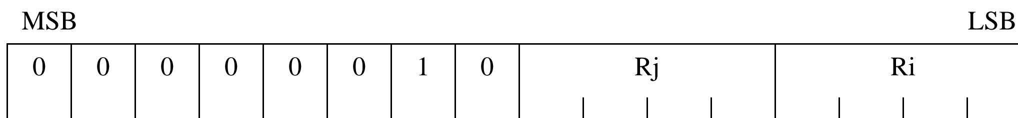
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

b cycle

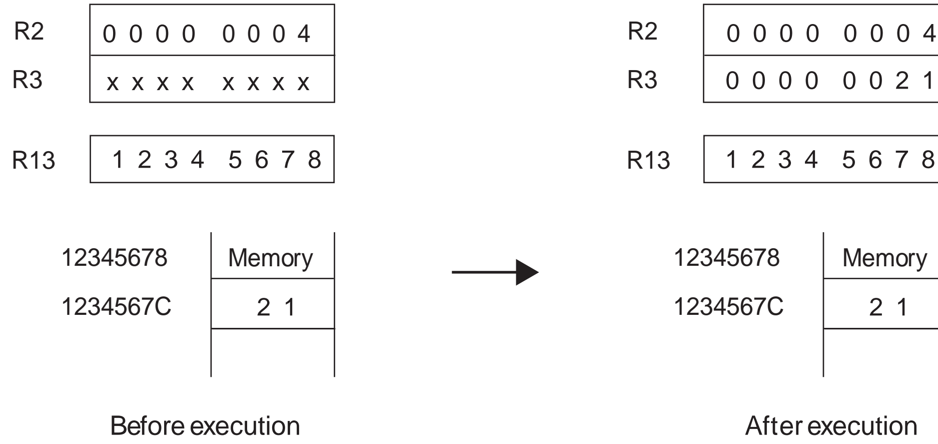
- Instruction Format



FR81 Family

- Execution Example

LDUB @(R13, R2), R3 ; Bit pattern of the instruction: 0000 0010 0010 0011



7.113 LDUB (Load Byte Data in Memory to Register)

Extends with zeros the byte data at memory address 14 + o8, loads to Ri. The value o8 is a signed calculation. The value of o8 is specified in disp8.

● Assembler Format

LDUB @(R14, disp8), Ri

● Operation

extu((R14+o8)) → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

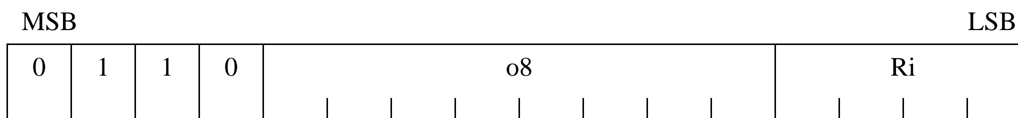
● Classification

Memory Load instruction, Instruction with delay slot

● Execution Cycles

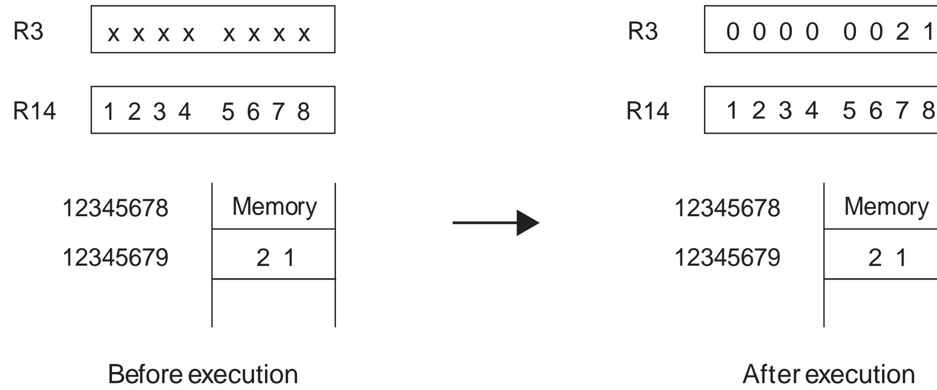
b cycle

● Instruction Format



● Execution Example

LDUB @(R14,1), R3 ; Bit pattern of the instruction: 0110 0000 0001 0011



7.114 LDUB (Load Byte Data in Memory to Register)

Loads the byte data at memory address BP+u16 to Ri. Unsigned u16 value is calculated. The value in u16 is specified as udisp16.

- Assembler Format

LDUB @(BP, udisp16), Ri

- Operation

(BP+u16) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

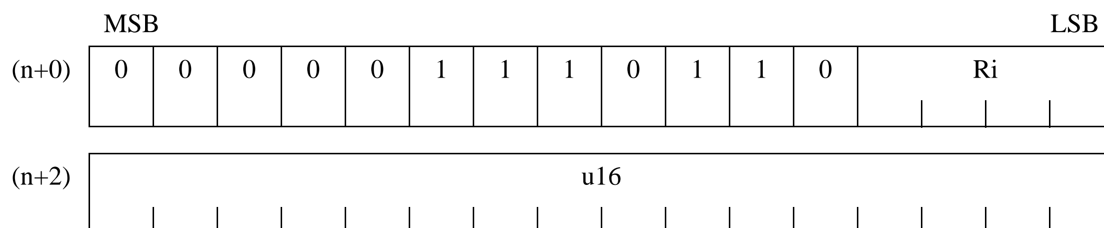
- Classification

Memory Load instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

7.115 LDUH (Load Halfword Data in Memory to Register)

Extends with zeros the half-word data at memory address Rj, loads to Ri.

● Assembler Format

LDUH @Rj, Ri

● Operation

extu((Rj)) → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

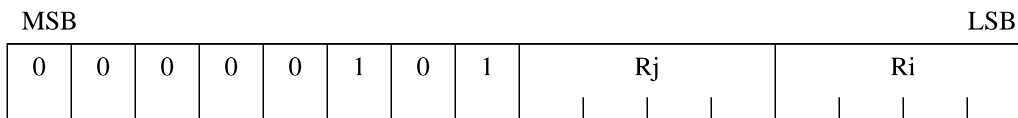
● Classification

Memory Load instruction, Instruction with delay slot

● Execution Cycles

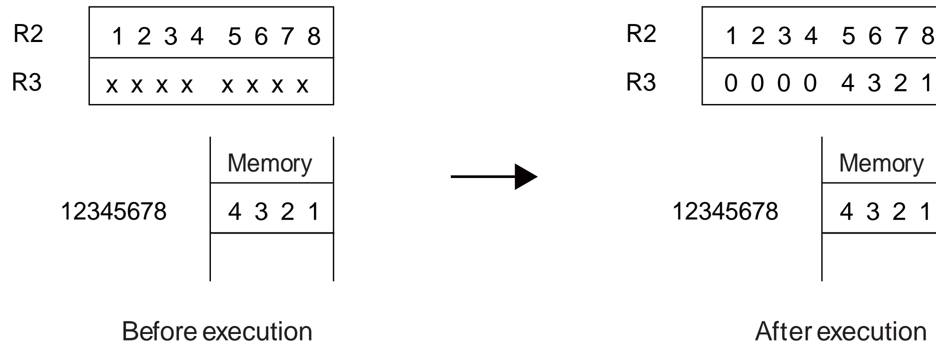
b cycle

● Instruction Format



- Execution Example

LDUH @R2, R3 ; Bit pattern of the instruction: 0000 0101 0010 0011



FR81 Family

7.116 LDUH (Load Halfword Data in Memory to Register)

Extends with zeros the half-word data at memory address R13 + Rj, loads to Ri.

- Assembler Format

LDUH @(R13, Rj), Ri

- Operation

extu((R13+Rj)) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

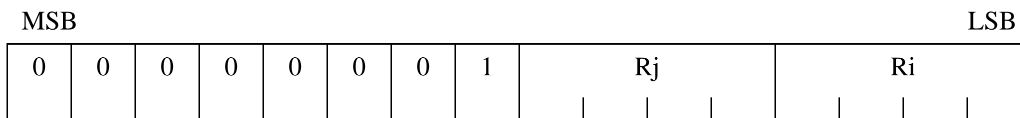
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

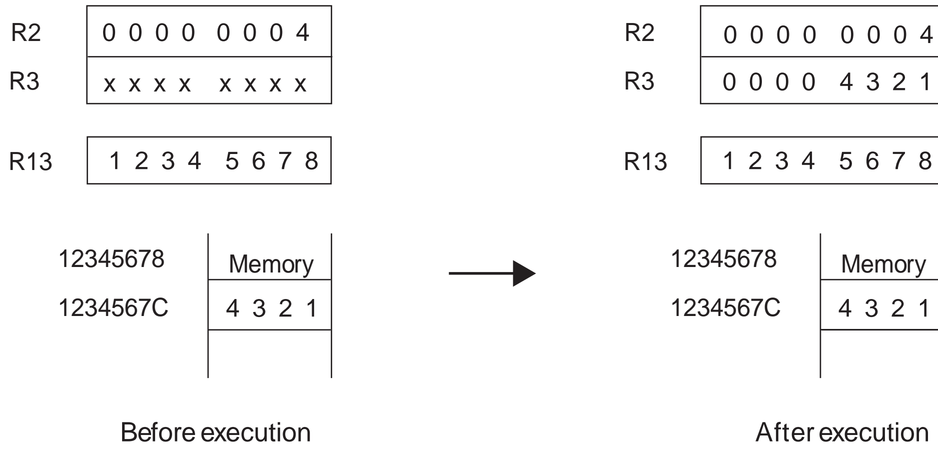
b cycle

- Instruction Format



- Execution Example

LDUH @(R13, R2), R3 ; Bit pattern of the instruction: 0000 0001 0010 0011



7.117 LDUH (Load Halfword Data in Memory to Register)

Extends with zeros the half-word data at memory address R14 + o8 × 2, loads to Ri. The value o8 is a signed calculation. The value of o8 × 2 is specified in disp9.

- Assembler Format

LDUH @(R14, disp9), Ri

- Operation

extu((R14+o8 × 2)) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

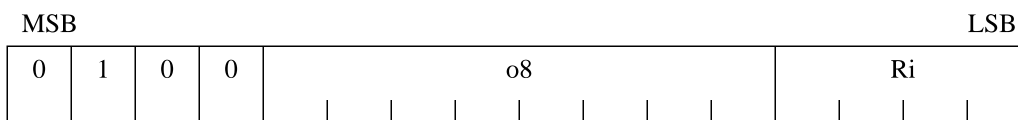
- Classification

Memory Load instruction, Instruction with delay slot

- Execution Cycles

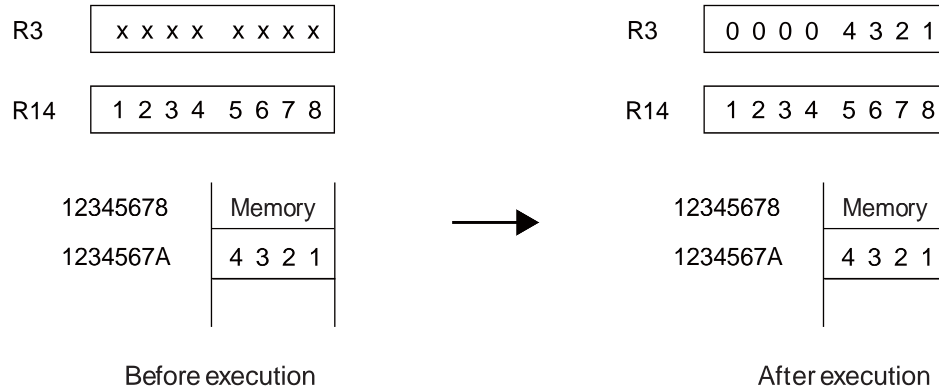
b cycle

- Instruction Format



- Execution Example

LDUH @(R14,2), R3 ; Bit pattern of the instruction: 0100 0000 0001 0011



FR81 Family

7.118 LDUH (Load Halfword Data in Memory to Register)

Loads the half word data at memory address BP+u16 × 2 to Ri. Unsigned u16 value is calculated. The value in u16 × 2 is specified as udisp17.

- Assembler Format

LD @(BP, udisp17), Ri

- Operation

(BP+u16 × 2) → Ri

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

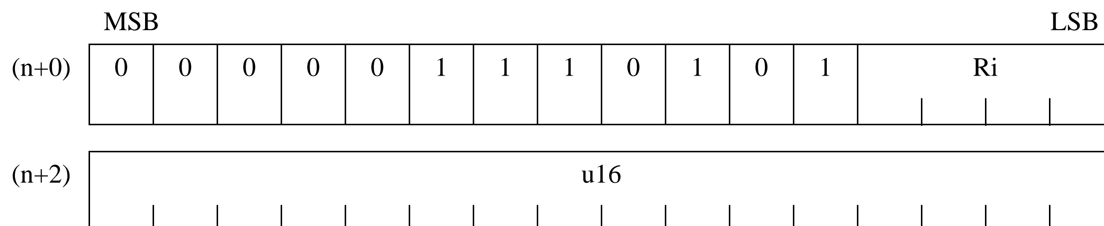
- Classification

Memory Load instruction, FR81 family

- Execution Cycles

a cycle

- Instruction Format



- EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

7.119 LEAVE (Leave Function)

This instruction is used for stack frame release processing for high level languages.

- Assembler Format

LEAVE

- Operation

R14+4 → R15

(R15-4) → R14

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Other instructions, Instruction with delay slot

- Execution Cycles

b cycle

- Instruction Format

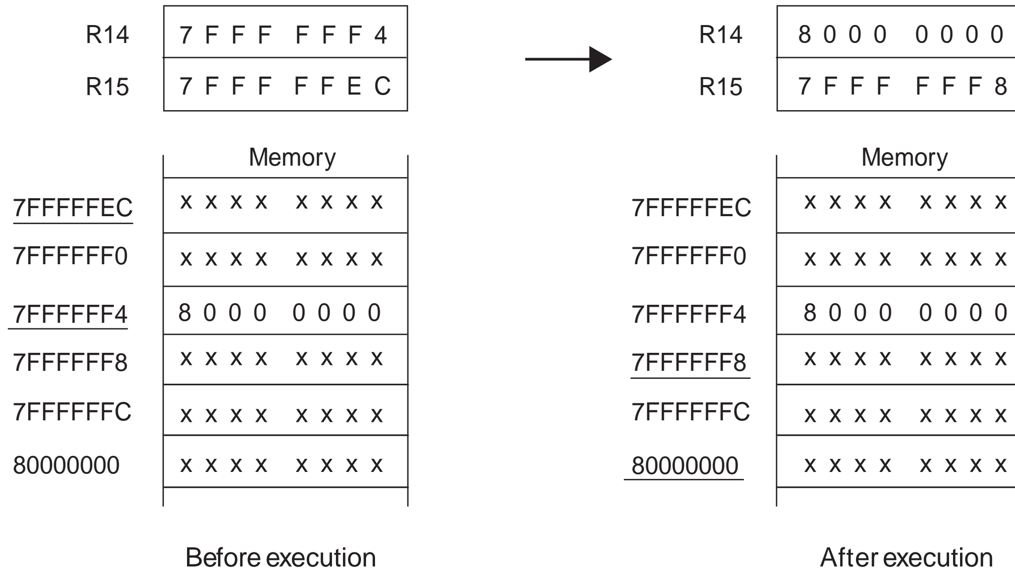
MSB LSB

1	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0

FR81 Family

- Execution Example

LEAVE ; Bit pattern of the instruction: 1001 1111 1001 0000



7.120 LSL (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in Ri by Rj bits, stores the result to Ri. Only the lower 5 bits of Rj, which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

● Assembler Format

LSL Rj, Ri

● Operation

Ri << Rj → Ri

● Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last. Cleared when the shift amount is zero.

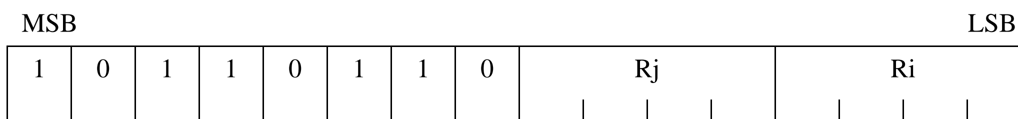
● Classification

Shift instruction, Instruction with delay slot

● Execution Cycles

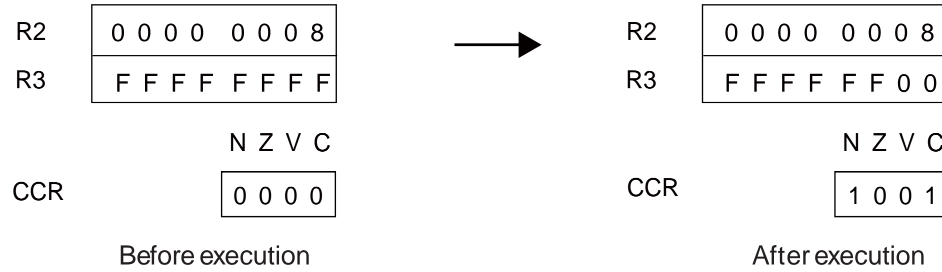
1 cycle

● Instruction Format



- Execution Example

LSL R2, R3 ; Bit pattern of the instruction: 1011 0110 0010 0011



7.121 LSL (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in Ri by u4 bits, stores the result to Ri.

- Assembler Format

LSL #u4, Ri

- Operation

Ri << u4 → Ri

- Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last. Cleared when the shift amount is zero.

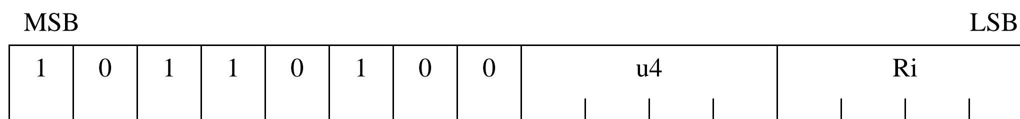
- Classification

Shift instruction, Instruction with delay slot

- Execution Cycles

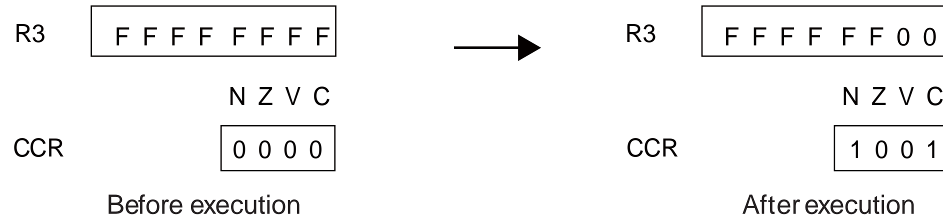
1 cycle

- Instruction Format



- Execution Example

LSL #8, R3 ; Bit pattern of the instruction: 1011 0100 1000 0011



7.122 LSL2 (Logical Shift to the Left Direction)

Makes a logical left shift of the word data in Ri by u4+16 bits, stores the result to Ri.

- Assembler Format

LSL2 #u4, Ri

- Operation

Ri << {u4+16} → Ri

- Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last.

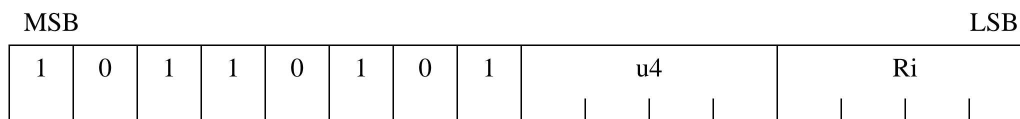
- Classification

Shift instruction, Instruction with delay slot

- Execution Cycles

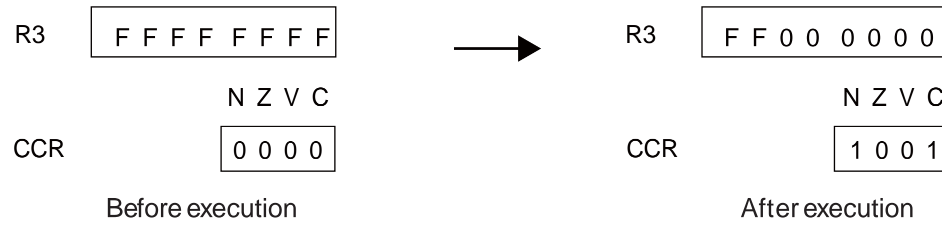
1 cycle

- Instruction Format



- Execution Example

LSL2 #8, R3 ; Bit pattern of the instruction: 1011 0101 1000 0011



7.123 LSR (Logical Shift to the Right Direction)

Makes a logical right shift of the word data in Ri by Rj bits, stores the result to Ri. Only the lower 5 bits of Rj, which designates the size of the shift, are valid and the shift range is 0 to 31 bits.

● Assembler Format

LSR Rj, Ri

● Operation

Ri >> Rj → Ri

● Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last. Cleared when the shift amount is zero.

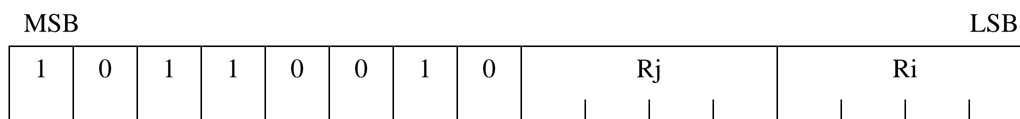
● Classification

Shift instruction, Instruction with delay slot

● Execution Cycles

1 cycle

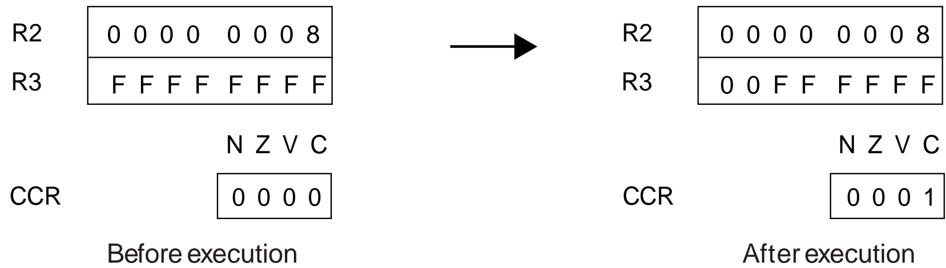
● Instruction Format



FR81 Family

● Execution Example

LSR R2, R3 ; Bit pattern of the instruction: 1011 0010 0010 0011



7.124 LSR (Logical Shift to the Right Direction)

Makes a logical left shift of the word data in Ri by u4 bits, stores the result to Ri.

- Assembler Format

LSR #u4, Ri

- Operation

Ri >> u4 → Ri

- Flag Change

N	Z	V	C
C	C	-	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last. Cleared when the shift amount is zero.

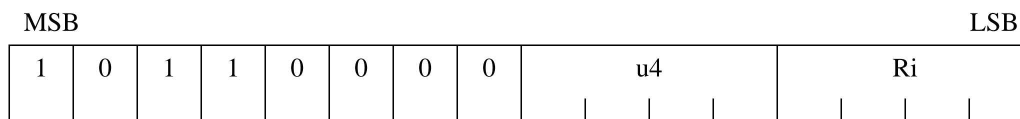
- Classification

Shift instruction, Instruction with delay slot

- Execution Cycles

1 cycle

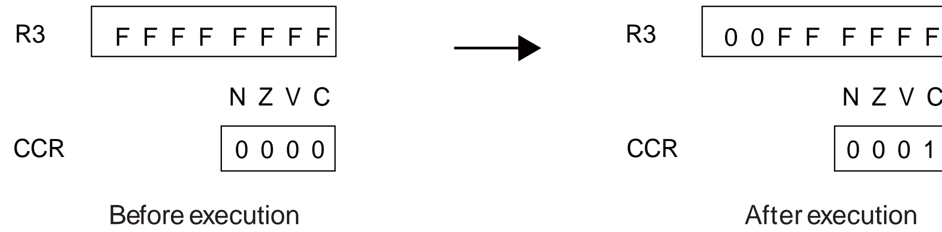
- Instruction Format



FR81 Family

- Execution Example

LSR #8, R3 ; Bit pattern of the instruction: 1011 0000 1000 0011



7.125 LSR2 (Logical Shift to the Right Direction)

Makes a logical left shift of the word data in Ri by u4+16 bits, stores the result to Ri.

- Assembler Format

LSR2 #u4, Ri

- Operation

Ri >> {u4+16} → Ri

- Flag Change

N	Z	V	C
C	C	-	C

N: Cleared.

Z: Set when the operation result is zero, cleared otherwise.

V: Unchanged.

C: Holds the bit value shifted last.

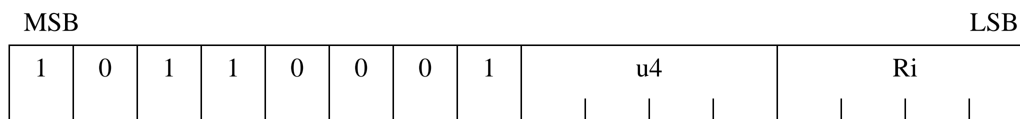
- Classification

Shift instruction, Instruction with delay slot

- Execution Cycles

1 cycle

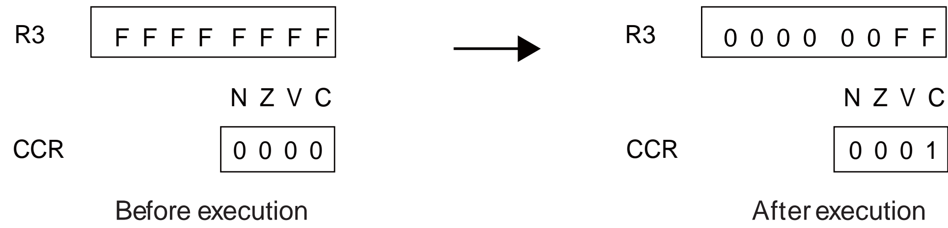
- Instruction Format



FR81 Family

- Execution Example

LSR2 #8, R3 ; Bit pattern of the instruction: 1011 0001 1000 0011



7.126 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in Rj to Ri.

● Assembler Format

MOV Rj, Ri

● Operation

Rj → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

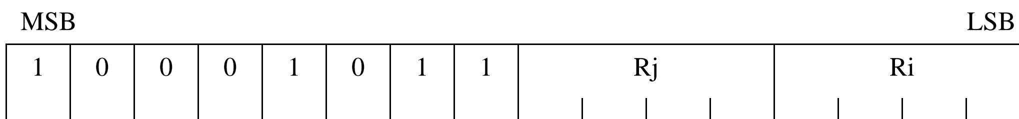
● Classification

Inter-register transfer instruction, Instruction with delay slot

● Execution Cycles

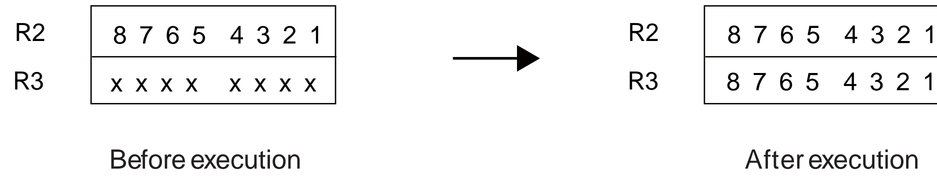
1 cycle

● Instruction Format



● Execution Example

MOV R2, R3 ; Bit pattern of the instruction: 1000 1011 0010 0011



7.127 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data in dedicated register Rs to general-purpose register Ri.

● Assembler Format

MOV Rs, Ri

● Operation

Rs → Ri

If the number of a non-existent dedicated register is given as Rs, undefined data will be transferred.

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

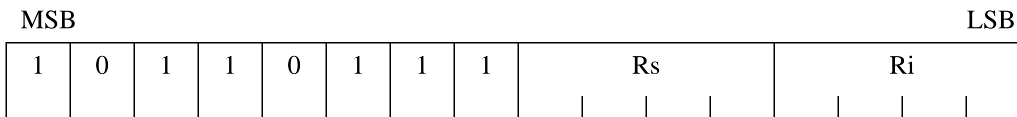
● Classification

Inter-Register Transfer instruction, Instruction with delay slot

● Execution Cycles

1 cycle

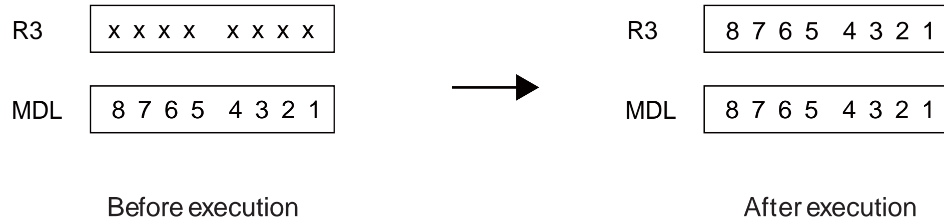
● Instruction Format



FR81 Family

- Execution Example

MOV MDL, R3 ; Bit pattern of the instruction: 1011 0111 0101 0011



7.128 MOV (Move Word Data in Program Status Register to Destination Register)

Moves the word data in the program status (PS) to general-purpose register Ri.

● Assembler Format

MOV PS, Ri

● Operation

PS → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

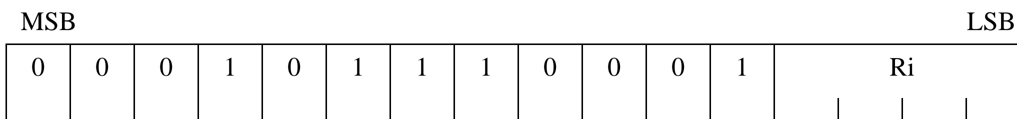
● Classification

Inter-Register Transfer instruction, Instruction with delay slot

● Execution Cycles

1 cycle

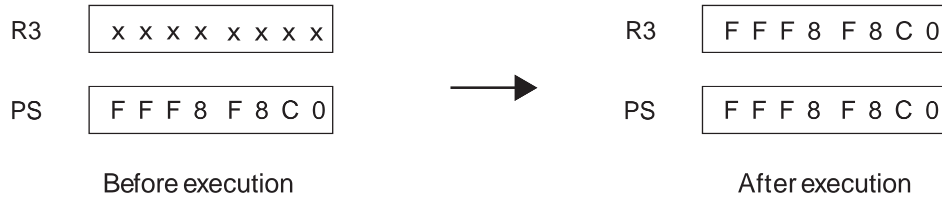
● Instruction Format



FR81 Family

- Execution Example

MOV PS, R3 ; Bit pattern of the instruction: 0001 0111 0001 0011



7.129 MOV (Move Word Data in Source Register to Destination Register)

Moves the word data of general-purpose register Ri to dedicated register Rs.

- Assembler Format

MOV Ri, Rs

- Operation

Ri → Rs

If TBR, SSP, or ESR is specified in user mode or if a number without a dedicated register is specified for "RS", it generates an invalid instruction exception (system-only register access).

There is no restriction in privilege mode. If the number of a non-existent register is given as parameter Rs in privilege mode, the read value Ri will be ignored.

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

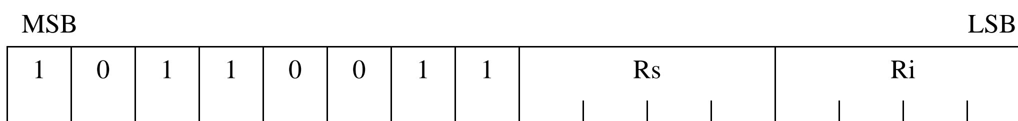
- Classification

Inter-Register Transfer instruction, Instruction with delay slot, FR81 updating

- Execution Cycles

1 cycle

- Instruction Format



● EIT Occurrence and Detection

User mode:

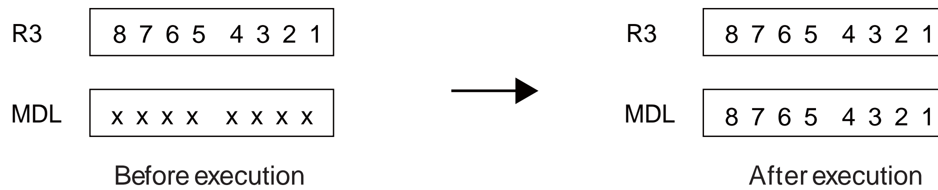
An invalid instruction exception (system-only register access) is generated and an interrupt is detected.

Privilege mode:

An interrupt is detected.

● Execution Example

MOV R3, MDL ; Bit pattern of the instruction: 1011 0011 0101 0011



7.130 MOV (Move Word Data in Source Register to Program Status Register)

Stores the word data of general-purpose register Ri to program status (PS).

● Assembler Format

MOV Ri, PS

● Operation

Ri → PS

The contents of system status register (SSR) cannot be changed by this instruction regardless of the selected operation mode. If this instruction is executed in user mode, only the D1, D0, N, Z, V, and C flags can be changed. The other flag values are not updated. Any bit other than SSR can be changed in privilege mode.

At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded from Ri, the value +16 is transferred to the ILM. If the original ILM value is in the range 0 to 15, then any value from 0 to 31 can be transferred to the ILM.

● Flag Change

N	Z	V	C
C	C	C	C

N, Z, V, C: Data from Ri is transferred.

● Classification

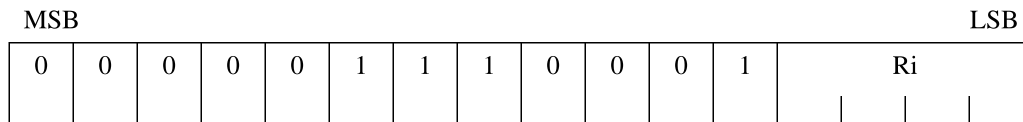
Inter-Register Transfer instruction, Instruction with delay slot, FR81 updating

● Execution Cycles

1 cycle

FR81 Family

● Instruction Format

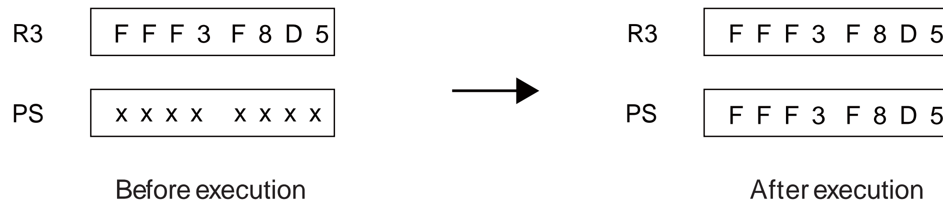


● EIT Occurrence and Detection

An interrupt is detected.

● Execution Example

MOV R3, PS ; Bit pattern of the instruction: 0000 0111 0001 0011



7.131 MOV (Move Word Data in General Purpose Register to Floating Point Register)

The value in Rj is transferred to FRi.

- Assembler Format

MOV Rj, FRi

- Operation

Rj → FRi

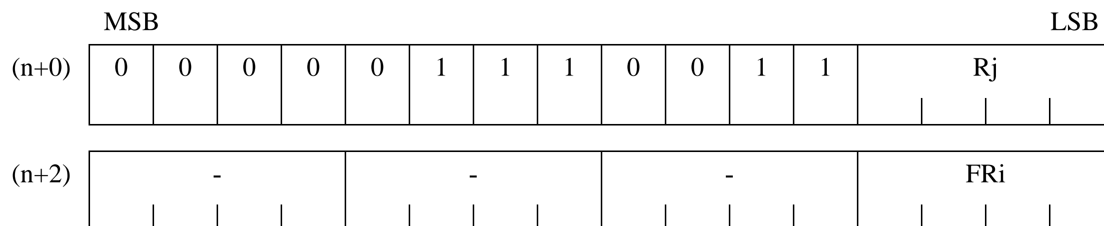
- Classification

Single-precision floating point instruction, FR81 family

- Execution Cycles

1 cycle

- Instruction Format



- EIT Occurrence and Detection

An invalid instruction exception (FPU absence error) or an interrupt is detected.

FR81 Family

7.132 MOV (Move Word Data in Floating Point Register to General Purpose Register)

The value in FRi is transferred to Rj.

● Assembler Format

MOV FRi, Rj

● Operation

FRi → Rj

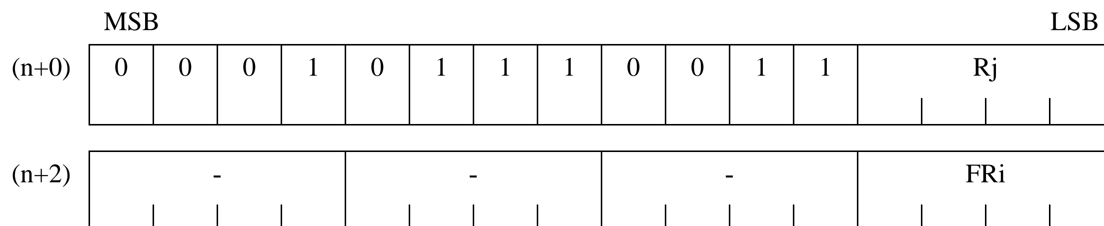
● Classification

Single-precision floating point instruction, FR81 family

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

An invalid instruction exception (FPU absence error) or an interrupt is detected.

7.133 MUL (Multiply Word Data)

Multiplies the word data in Rj by the word data in Ri as signed numbers, and stores the resulting signed 64-bit data with the higher word in the multiplication/division register (MDH), and the lower word in the multiplication/division register (MDL).

● Assembler Format

MUL Rj, Ri

● Operation

Ri × Rj → MDH, MDL

● Flag Change

N	Z	V	C
C	C	C	-

N: Set when the MSB of the "MDL" of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Cleared when the operation result is in the range -2147483648 to 2147483647, cleared otherwise.

C: Unchanged.

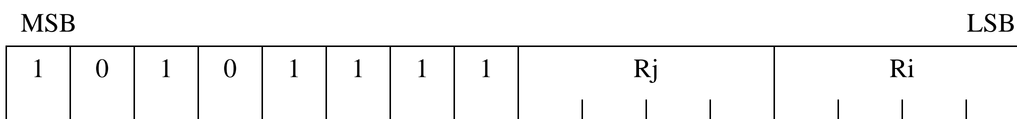
● Classification

Multiply/Divide Instruction

● Execution Cycles

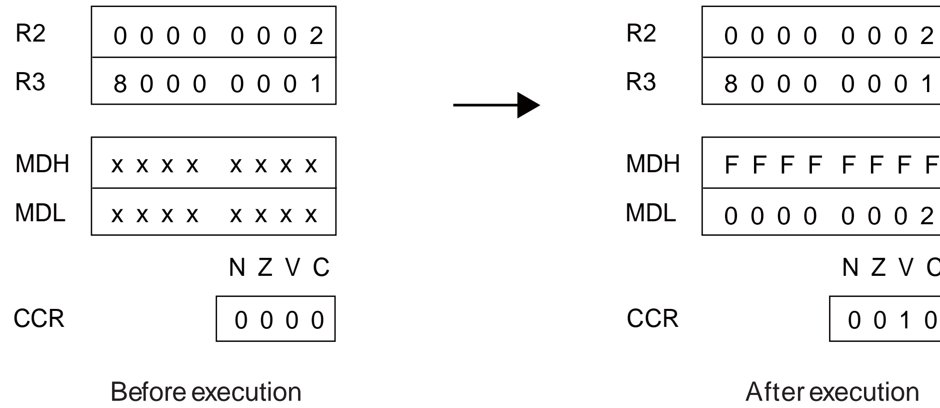
5 cycles

● Instruction Format



- Execution Example

MUL R2, R3 ; Bit pattern of the instruction: 1010 1111 0010 0011



7.134 MULH (Multiply Halfword Data)

Multiplies the half-word data in the lower 16 bits of Rj by the half-word data in the lower 16 bits of Ri as signed numbers, and stores the resulting signed 32-bit data in the multiplication/division register (MDL). The multiplication/division register (MDH) is undefined.

- Assembler Format

MULH Rj, Ri

- Operation

Ri × Rj → MDL

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the MDL of the operation result is "1", cleared when the MSB is "0".

Z: Set when MDL of the operation result is zero, cleared otherwise.

V, C: Unchanged.

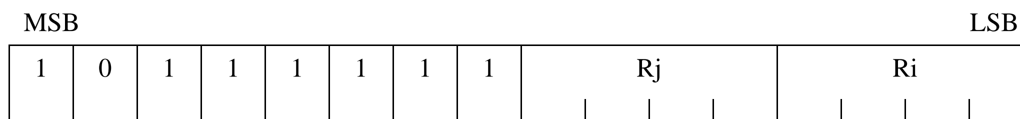
- Classification

Multiply/Divide Instruction

- Execution Cycles

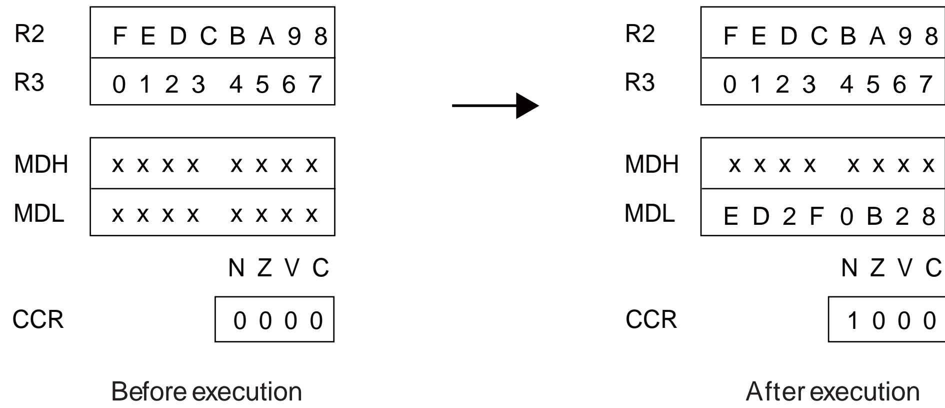
3 cycles

- Instruction Format



- Execution Example

MULH R2, R3 ; Bit pattern of the instruction: 1011 1111 0010 0011



7.135 MULU (Multiply Unsigned Word Data)

Multiplies the word data in Rj by the word data in Ri as unsigned numbers and stores the resulting unsigned 64-bit data with the higher word in the multiplication/division register (MDH), and the lower word in the multiplication/division register (MDL).

- Assembler Format

MULU Rj, Ri

- Operation

Ri × Rj → MDH, MDL

- Flag Change

N	Z	V	C
C	C	C	-

N: Set when the MSB of the MDL of the operation result is "1", cleared when the MSB is "0".

Z: Set when the MDL of the operation result is zero, cleared otherwise.

V: Cleared when the operation result is in the range 0 to 4294967295, set otherwise.

C: Unchanged.

- Classification

Multiply/Divide Instruction

- Execution Cycles

5 cycles

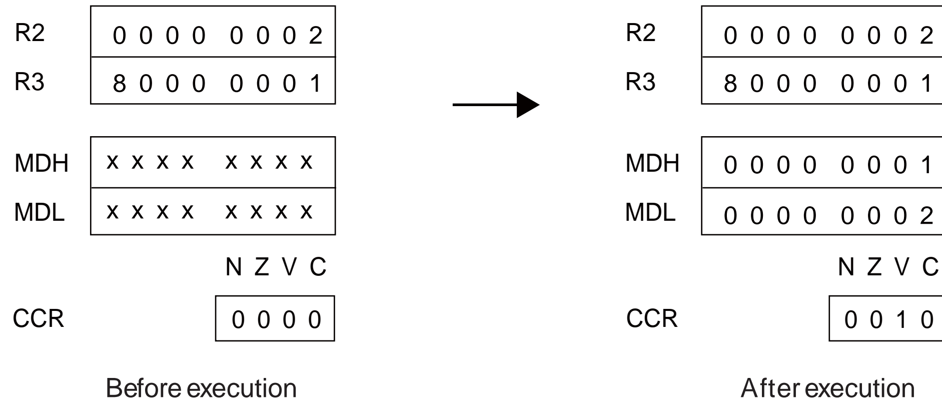
- Instruction Format

MSB ... LSB

1	0	1	0	1	0	1	1	Rj	Ri

- Execution Example

MULU R2, R3 ; Bit pattern of the instruction: 1010 1011 0010 0011



7.136 MULUH (Multiply Unsigned Halfword Data)

Multiplies the half-word data in the lower 16 bits of Rj by the half-word data in the lower 16 bits of Ri as unsigned numbers, and stores the resulting unsigned 32-bit data in the multiplication/division register (MDL). The multiplication/division register (MDH) is undefined.

● Assembler Format

MULUH Rj, Ri

● Operation

Ri × Rj → MDL

● Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the MDL of the operation result is "1", cleared when the MSB is "0".

Z: Set when the MDL of the operation result is zero, cleared otherwise.

V, C: Unchanged.

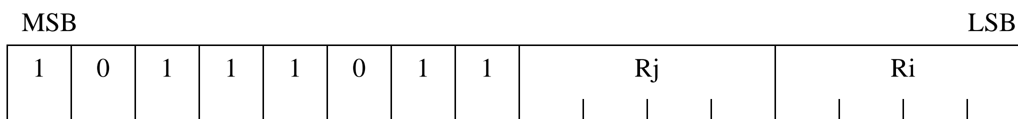
● Classification

Multiply/Divide Instruction

● Execution Cycles

3 cycles

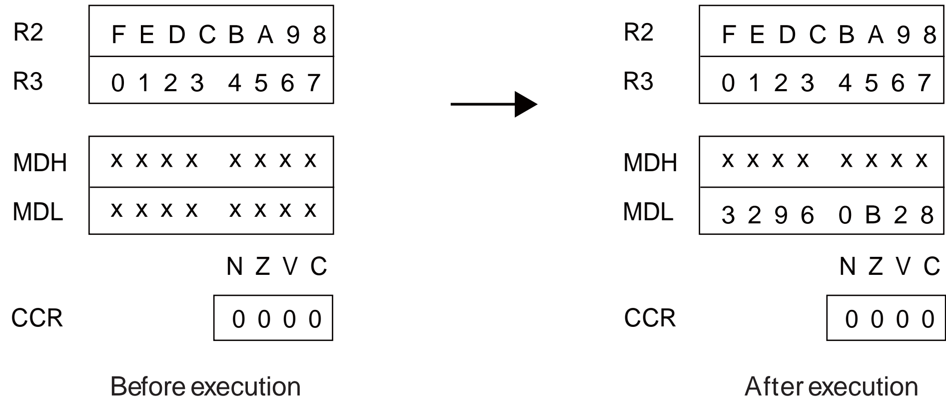
● Instruction Format



FR81 Family

● Execution Example

MULUH R2, R3 ; Bit pattern of the instruction: 1011 1011 0010 0011



7.137 NOP (No Operation)

This instruction performs no operation.

● Assembler Format

NOP

● Operation

No operation

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

Other instructions, Instruction with delay slot

● Execution Cycles

1 cycle

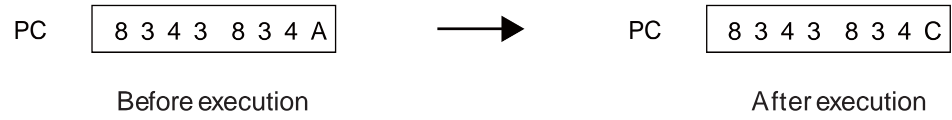
● Instruction Format

MSB LSB

1	0	0	1	1	1	1	1	1	0	1	0	0	0	0	0

● Execution Example

NOP ; Bit pattern of the instruction: 1001 1111 1010 0000



7.138 OR (Or Word Data of Source Register to Data in Memory)

Takes the logical OR of the word data at memory address Ri and the word data in Rj, stores the results to the memory address corresponding to Ri.

- Assembler Format

OR Rj,@Ri

- Operation

(Ri) | Rj → (Ri)

- Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

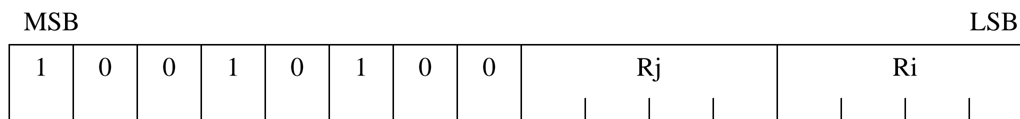
- Classification

Logical Calculation instruction, Read/Modify/Write type instruction

- Execution Cycles

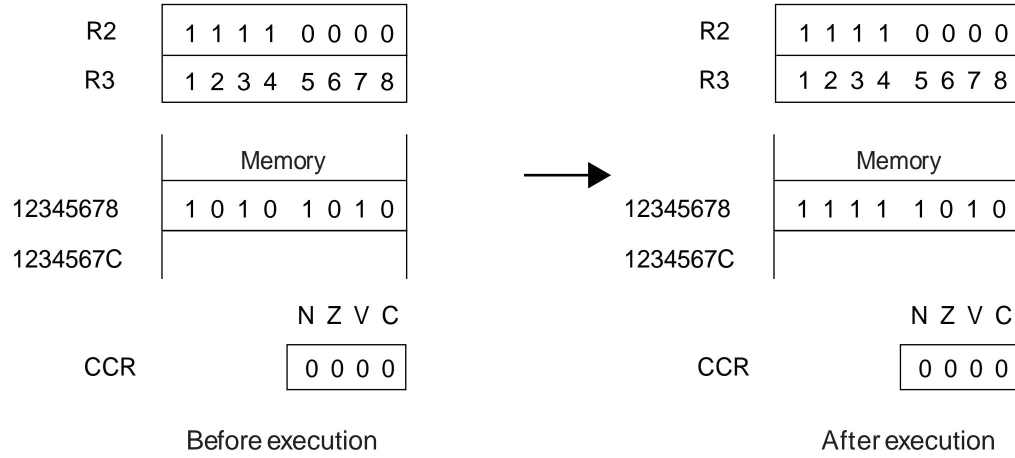
1+2a cycles

- Instruction Format



- Execution Example

```
OR   R2,@R3          ; Bit pattern of the instruction: 1001 0100 0010 0011
```



7.139 OR (Or Word Data of Source Register to Destination Register)

Takes the logical OR of the word data in Ri and the word data in Rj, stores the results to Ri.

● Assembler Format

OR Rj, Ri

● Operation

Ri | Rj → Ri

● Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

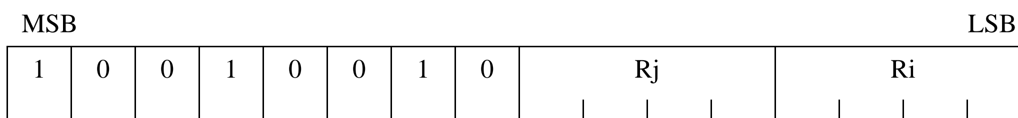
● Classification

Logical Calculation instruction, Instruction with delay slot

● Execution Cycles

1 cycle

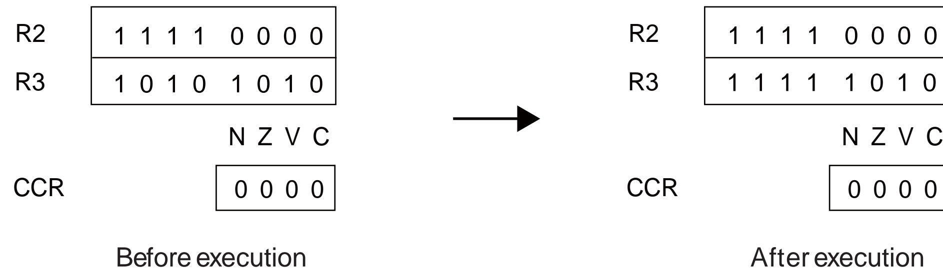
● Instruction Format



FR81 Family

- Execution Example

OR R2, R3 ; Bit pattern of the instruction: 1001 0010 0010 0011



7.140 ORB (Or Byte Data of Source Register to Data in Memory)

Takes the logical OR of the byte data at memory address Ri and the byte data in Rj, stores the results to the memory address corresponding to Ri.

● Assembler Format

ORB Rj,@Ri

● Operation

(Ri) | Rj → (Ri)

● Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB(bit7) of the operation result is "1", cleared when the MSB(bit7) is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

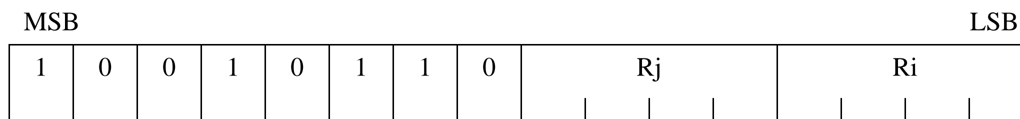
● Classification

Logical Calculation instruction, Read/Modify/Write type instruction

● Execution Cycles

1+2a cycles

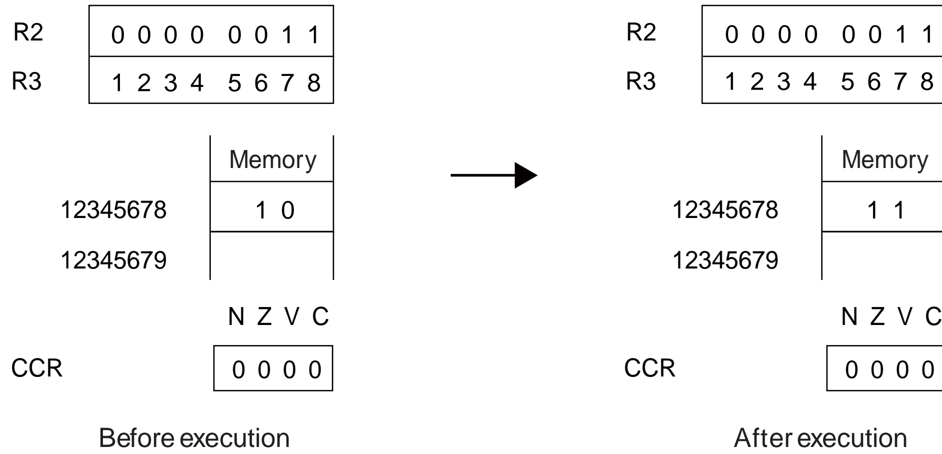
● Instruction Format



FR81 Family

- Execution Example

ORB R2,@R3 ; Bit pattern of the instruction: 1001 0110 0010 0011



7.141 ORCCR (Or Condition Code Register and Immediate Data)

Takes the logical OR of the byte data in the condition code register (CCR) and the immediate data, and returns the results in to the CCR.

● Assembler Format

ORCCR #u8

● Operation

User mode:

CCR | (u8 & CF_H) → CCR

Privilege mode:

CCR | u8 → CCR

In user mode, a request to rewrite the stack flag (S) or the interrupt enable flag (I) is ignored. The S and I flags can only be changed in privilege mode.

● Flag Change

S	I	N	Z	V	C
C	C	C	C	C	C

S, I, N, Z, V, C: Varies according to results of calculation.

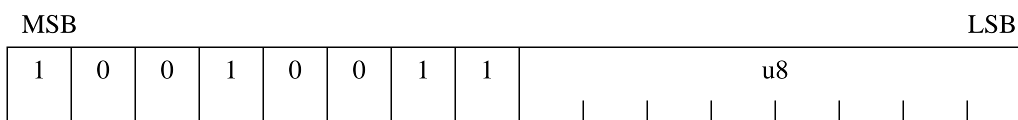
● Classification

Other instructions, Instruction with delay slot, FR81 updating

● Execution Cycles

1 cycle

● Instruction Format

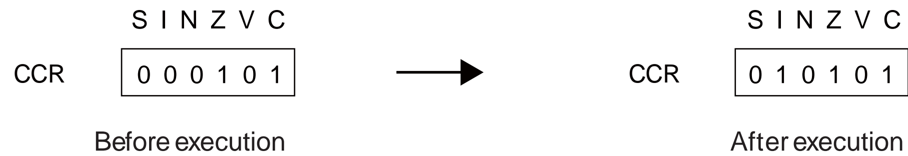


- EIT Occurrence and Detection

An interrupt is detected (the value of I flag after instruction execution is used).

- Execution Example

ORCCR #10H ; Bit pattern of the instruction: 1001 0011 0001 0000



7.142 ORH (Or Halfword Data of Source Register to Data in Memory)

Takes the logical OR if the half-word data at memory address Ri and the half-word data in Rj, stores the results to the memory address corresponding to Ri.

● Assembler Format

ORH Rj,@Ri

● Operation

(Ri) | Rj → (Ri)

● Flag Change

N	Z	V	C
C	C	-	-

N: Set when the MSB(bit15) of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V, C: Unchanged.

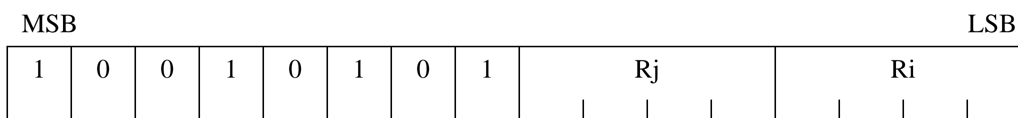
● Classification

Logical Calculation instruction, Read/Modify/Write type instruction

● Execution Cycles

1+2a cycles

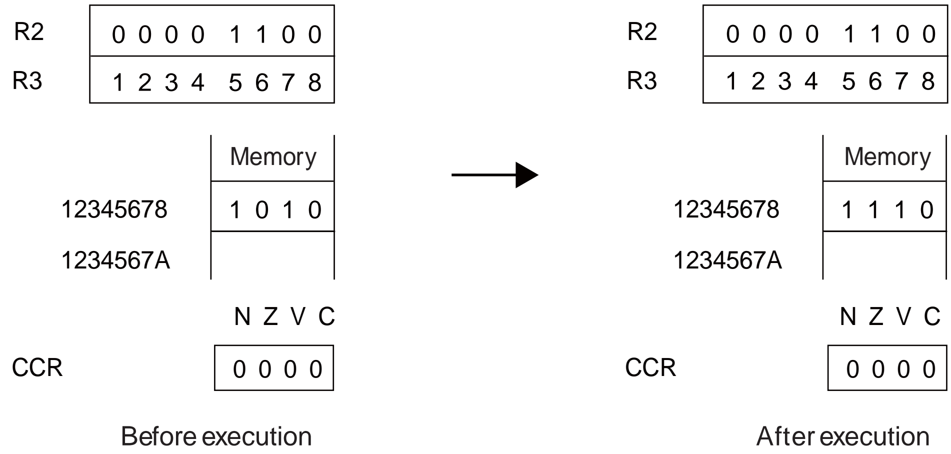
● Instruction Format



FR81 Family

● Execution Example

ORH   R2,@R3   ; Bit pattern of the instruction: 1001 0101 0010 0011



7.143 RET (Return from Subroutine)

This is a branching instruction without a delay slot. Branches to the address indicated by the return pointer(RP). Used for return from Subroutine.

- Assembler Format

RET

- Operation

RP → PC

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Non-Delayed Branching instruction

- Execution Cycles

2 cycles

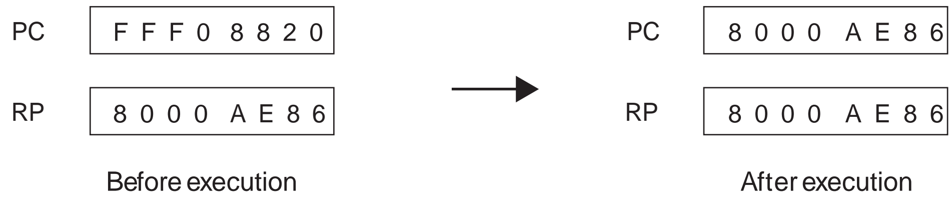
- Instruction Format

MSB LSB

1	0	0	1	0	1	1	1	0	0	1	0	0	0	0	0

- Execution Example

RET ; Bit pattern of the instruction: 1001 0111 0010 0000



7.144 RET:D (Return from Subroutine)

This is a branching instruction with a delay slot. Branches to the address indicated by the return pointer (RP). Used for return from Subroutine.

- Assembler Format

RET:D

- Operation

RP → PC

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Delayed Branching instruction

- Execution Cycles

1 cycle

- Instruction Format

MSB ... LSB

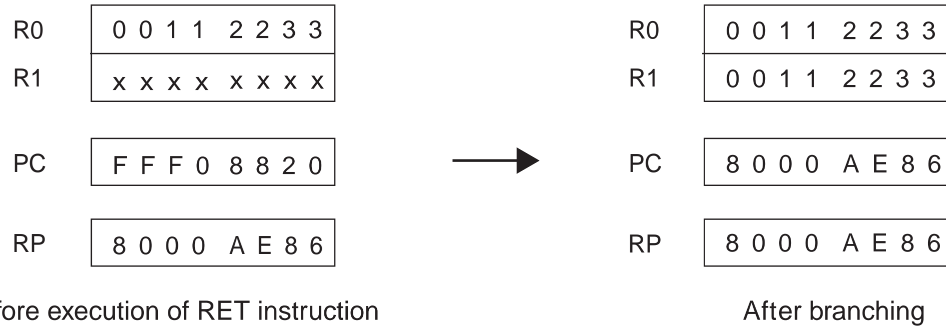
1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0

FR81 Family

- Execution Example

RET:D ; Bit pattern of the instruction: 1001 1111 0010 0000

MOV R0, R1 ; Instruction placed in delay slot



Before execution of RET instruction

After branching

The instruction placed in delay slot will be executed before execution of the branch destination instruction. The value of R1 above will vary according to the specifications of the MOV instruction placed in the delay slot.

7.145 RETI (Return from Interrupt)

Loads data from the stack indicated by system stack pointer (SSP), to the program counter (PC) and program status (PS), and retakes control from the EIT operation handler.

● Assembler Format

RETI

● Operation

- Normal operation state
  (SSP) → PC
  SSP+4 → SSP
  (SSP) → PS
  SSP+4 → SSP
- Debug state
  PC save register (PCSR) → PC
  PS save register (PSSR) → PS
  (PC) instruction execution

This is a privilege instruction, which is only available in privilege mode. If this instruction is executed in user mode, it generates an invalid instruction exception (privilege instruction execution).

Operation varies depending on whether this instruction is executed in the normal operation state or debug state. If this instruction is executed in the debug state, the DSU register is used instead of a stack, and the acceptance of interrupts is withheld until the next instruction execution is completed. This, therefore, executes one instruction necessarily after the RETI instruction was executed in the debug state.

At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If data in the range 0 to 15 is loaded in memory, the value 16 will be added to that data before being transferred to the ILM. If the original ILM value is in the range 0 to 15, then any value between 0 and 31 can be transferred to the ILM.

● Flag Change

S	I	N	Z	V	C
C	C	C	C	C	C

D2, D1, S, I, N, Z, V, C: Change according to the values retrieved from the stack.

FR81 Family

● Classification

Non-Delayed Branching instruction, FR81 updating

● Execution Cycles

1+2b cycles

● Instruction Format

MSB LSB

1	0	0	1	0	1	1	1	0	0	1	1	0	0	0	0

● EIT Occurrence and Detection

User mode:

An invalid instruction exception (privilege instruction execution) is generated.

Privilege mode:

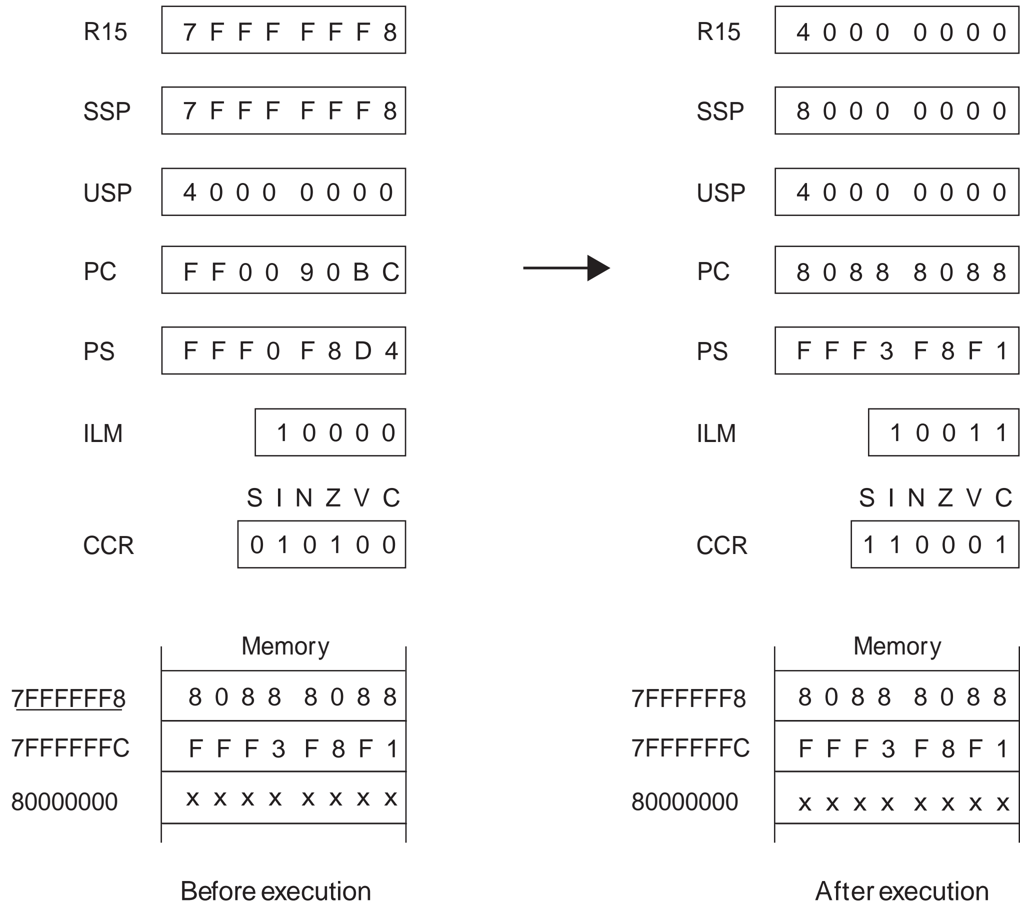
A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected. The interrupt level is judged using the value returned from the stack.

Debug state:

EIT is not accepted.

- Execution Example

RETI ; Bit pattern of the instruction: 1001 0111 0011 0000



FR81 Family

7.146 SRCH0 (Search First Zero bit position distance From MSB)

This is a "0" search instruction used for bit searching. Takes a comparison of word data in Ri from MSB (bit31) and "0", stores the distance in Ri from the first "0" that is found and bit MSB (bit31).

- Assembler Format

SRCH0 Ri

- Operation

search_zero(Ri) → Ri

If "0" bit is not found (in case all word data of Ri is "1" bit), 32 is stored in Ri. In case MSB(bit31) is "0", zero is stored in Ri and 31 is stored in Ri when LSB (bit0) is "0" and other bits are "1".

The Ri bit pattern before execution of instruction its relation with the values stored in Ri is shown in Table 7.146-1.

Table 7.146-1 Input pattern of SRCH0 instruction and its results

Input (Ri bit pattern before execution of instruction)	Result	Remarks
11111111_11111111_11111111_11111111	32	Not found
0xxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	0	MSB(bit31) was "1"
10xxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	1	
110xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	2	
1110xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	3	
. . .		
11111111_11111111_11111111_11110xxx	28	
11111111_11111111_11111111_111110xx	29	
11111111_11111111_11111111_1111110x	30	
11111111_11111111_11111111_11111110	31	Only LSB(bit0) was "0"

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

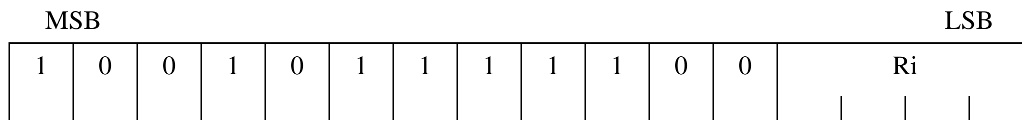
● Classification

Bit Search instruction, Instruction with delay slot

● Execution Cycles

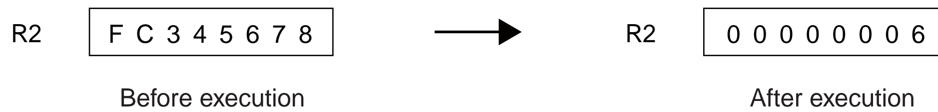
1 cycle

● Instruction Format



● Execution Example

SRCH0 R2 ; Bit pattern of the instruction: 1001 0111 1100 0010



FR81 Family

7.147 SRCH1 (Search First One bit position distance From MSB)

This is a "1" search instruction used for bit searching. Takes a comparison of word data in Ri from MSB (bit31) and "1", stores the distance in Ri from the first "1" that is found and bit MSB(bit31).

● Assembler Format

SRCH1 Ri

● Operation

search_one(Ri) $\rightarrow$ Ri

If "1" bit is not found (in case all word data of Ri is "0" bit), 32 is stored in Ri. In case MSB(bit31) is "1", zero is stored in Ri and 31 is stored in Ri when LSB (bit0) is "1" and other bits are "0".

The Ri bit pattern before execution of instruction its relation with the values stored in Ri is shown in Table 7.147-1.

Table 7.147-1 Input bit pattern of SRCH1 instruction and its results

Input (Ri bit pattern before execution of the instruction)	Results	Remarks
00000000_00000000_00000000_00000000	32	Not found
1xxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	0	MSB(bit31) was "0"
01xxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	1	
001xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	2	
0001xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	3	
. . .		
00000000_00000000_00000000_00001xxx	28	
00000000_00000000_00000000_000001xx	29	
00000000_00000000_00000000_0000001x	30	
00000000_00000000_00000000_00000001	31	Only LSB(bit0) was "0"

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

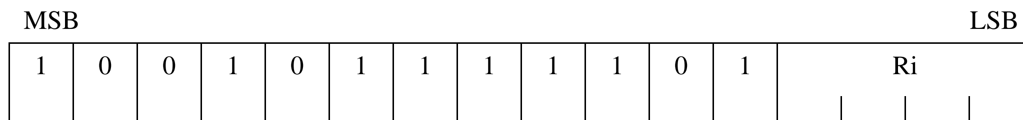
- Classification

Bit Search instruction, Instruction with delay slot

- Execution Cycles

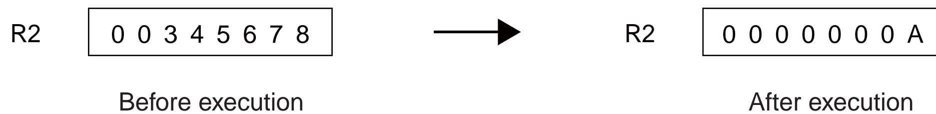
1 cycle

- Instruction Format



- Execution Example

SRCH0 R2 ; Bit pattern of the instruction: 1001 0111 1101 0010



7.148 SRCHC (Search First bit value change position distance From MSB)

This is a Change point search instruction used for bit searching. Takes a comparison of data in Ri with MSB (bit31), stores in Ri the distance from the first varying bit value that is found and bit MSB (bit31).

- Assembler Format

SRCHC Ri

- Operation

search_change(Ri) → Ri

If the values of all bits are the same, 32 is stored in Ri. In case the values of MSB (bit31) the adjacent bit30 is different, 1 is stored in Ri and 31 is stored in Ri when only the value of LSB (bit0) is different.

The Ri bit pattern before execution of instruction its relation with the values stored in Ri is shown in Table 7.148-1.

Table 7.148-1 Input bit pattern of SRCHC instruction and its results

Input (Ri bit pattern before execution of instruction)	Results	Remarks
00000000_00000000_00000000_00000000 11111111_11111111_11111111_11111111	32	Not found
01xxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx 10xxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	1	Difference between bit value of MSB(bit31) and bit30
001xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx 110xxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	2	
0001xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx 1110xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	3	
00001xxx_xxxxxxxx_xxxxxxxx_xxxxxxxx 11110xxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	4	
. . .		
00000000_00000000_00000000_00001xxx 11111111_11111111_11111111_11110xxx	28	
00000000_00000000_00000000_000001xx 11111111_11111111_11111111_111110xx	29	
00000000_00000000_00000000_0000001x 11111111_11111111_11111111_1111110x	30	
00000000_00000000_00000000_00000001 11111111_11111111_11111111_11111110	31	Bit value of only LSB(bit0) is different

* The value of result (Ri) can not become 0.

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

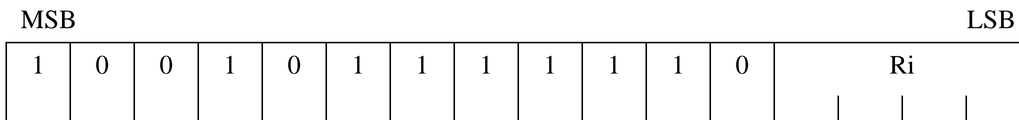
- Classification

Bit Search instruction, Instruction with delay slot

- Execution Cycles

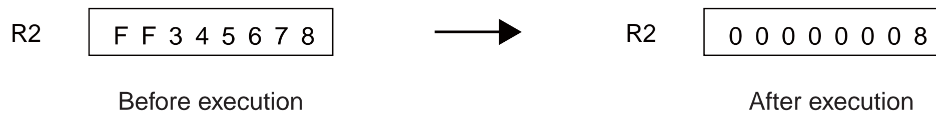
1 cycle

- Instruction Format



- Execution Example

SRCHC R2 ; Bit pattern of the instruction: 1001 0111 1110 0010



FR81 Family

7.149 ST (Store Word Data in Register to Memory)

Loads word data in Ri to memory address Rj.

● Assembler Format

ST Ri,@Rj

● Operation

Ri → (Rj)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

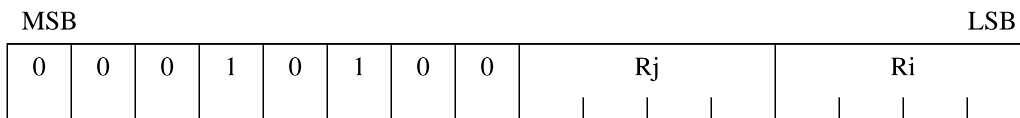
● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

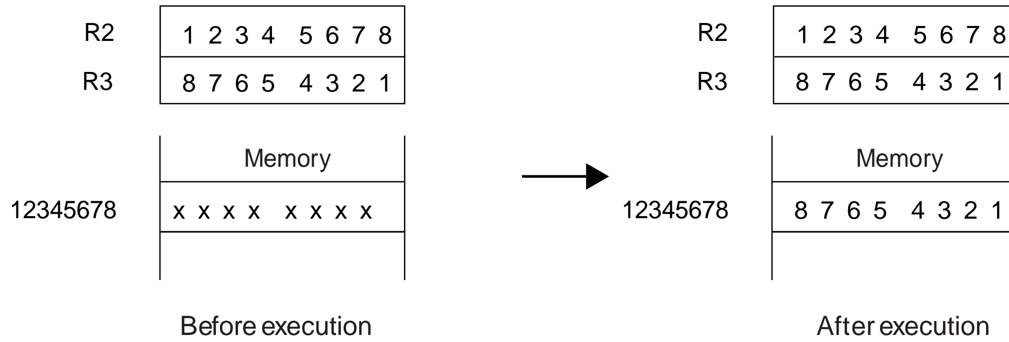
a cycle

● Instruction Format



- Execution Example

```
ST   R3,@R2          ; Bit pattern of the instruction: 0001 0100 0010 0011
```



FR81 Family

7.150 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address R13 + Rj.

● Assembler Format

ST Ri,@(R13, Rj)

● Operation

Ri → (R13+Rj)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

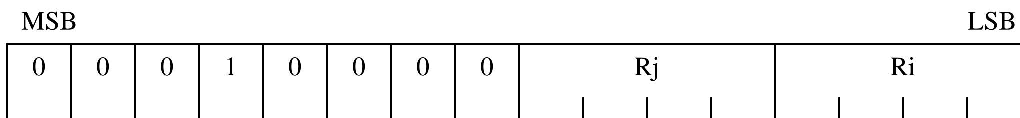
● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

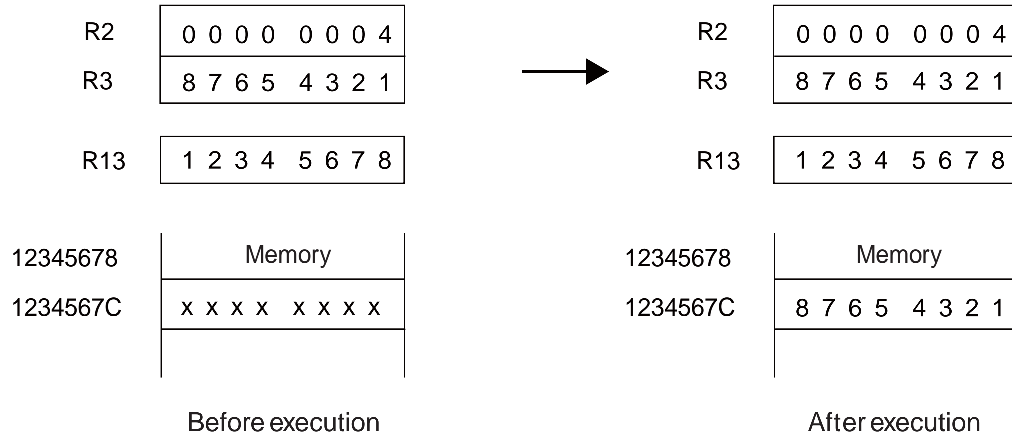
a cycle

● Instruction Format



● Execution Example

ST R3,@(R13, R2) ; Bit pattern of the instruction: 0001 0000 0010 0011



7.151 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address R14 + o8 × 4. The value o8 is a signed calculation. The value of o8 × 4 is specified in disp10.

- Assembler Format

ST Ri,@(R14, disp10)

- Operation

Ri → (R14+o8 × 4)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

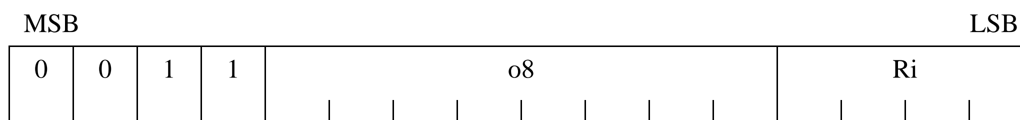
- Classification

Memory Store instruction, Instruction with delay slot

- Execution Cycles

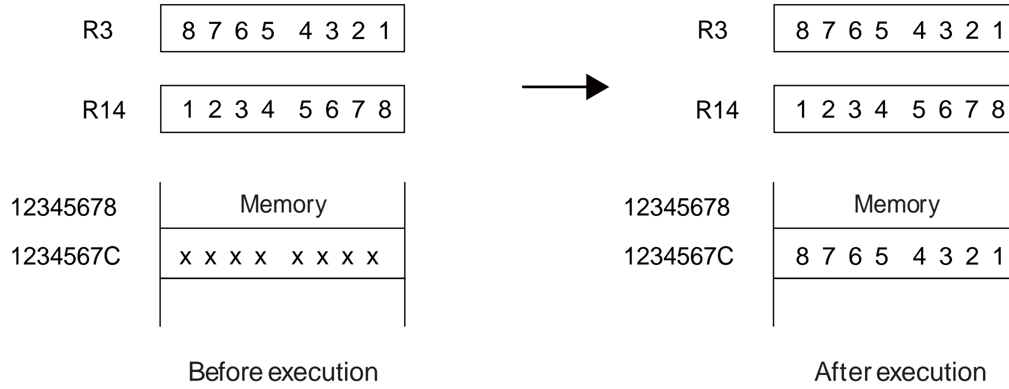
a cycle

- Instruction Format



● Execution Example

ST R3,@(R14,4) ; Bit pattern of the instruction: 0011 0000 0001 0011



FR81 Family

7.152 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address R15 + u4 × 4. The value u4 is an unsigned calculation. The value of u4 × 4 is specified in udisp6.

● Assembler Format

ST Ri,@(R15, udisp6)

● Operation

Ri → (R15+u4 × 4)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

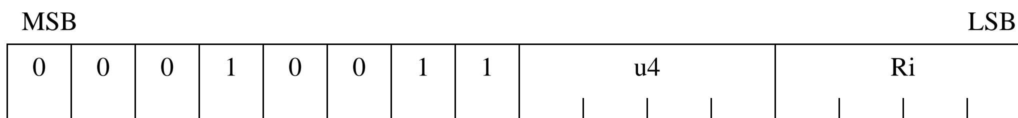
● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

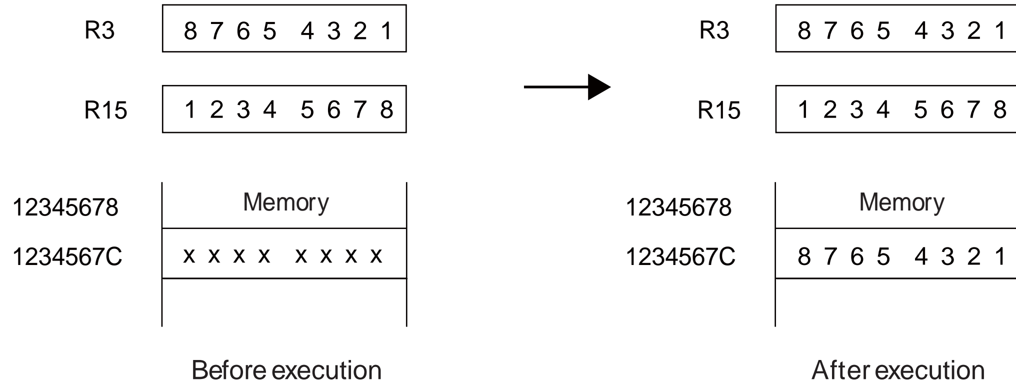
a cycle

● Instruction Format



- Execution Example

```
ST   R3,@(R15,4)    ; Bit pattern of the instruction: 0001 0011 0001 0011
```



7.153 ST (Store Word Data in Register to Memory)

Subtracts 4 from the value of R15, stores the word data in Ri to the memory address indicated by the new value of R15. If R15 is given as the parameter Ri, the data transfer will use the value of R15 before subtraction.

● Assembler Format

ST Ri,@-R15

● Operation

R15 - 4 → R15

Ri → (R15)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

a cycle

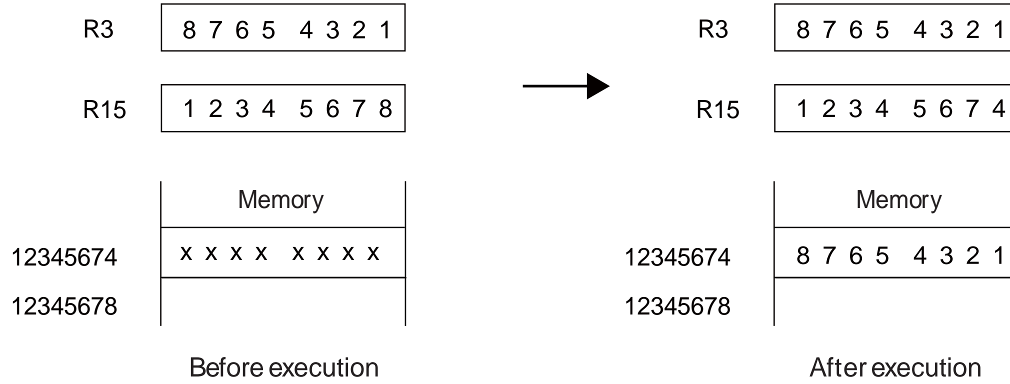
● Instruction Format

MSB ... LSB

0	0	0	1	0	1	1	1	0	0	0	0	Ri

- Execution Example

ST R3,@-R15 ; Bit pattern of the instruction: 0001 0111 0000 0011



FR81 Family

7.154 ST (Store Word Data in Register to Memory)

Loads the word data in Ri to memory address BP+u16 × 4. Unsigned u16 value is calculated. The value in u16 × 4 is specified as udisp18.

● Assembler Format

ST Ri, @(BP, udisp18)

● Operation

Ri → (BP+u16 × 4)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

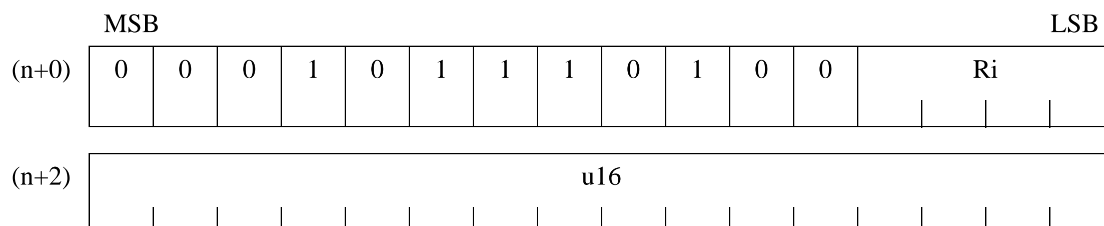
● Classification

Memory Store instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

7.155 ST (Store Word Data in Register to Memory)

Subtracts 4 from the value R15, stores the word data in dedicated register Rs to the memory address indicated by the new value of R15.

- Assembler Format

ST Rs,@-R15

- Operation

R15 - 4 → R15

Rs → (R15)

If the number of a non-existent dedicated register is specified in parameter Rs, the read value will be ignored.

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

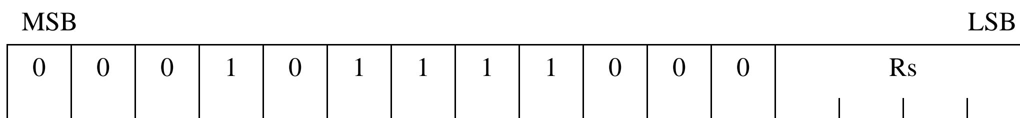
- Classification

Memory Store instruction, Instruction with delay slot

- Execution Cycles

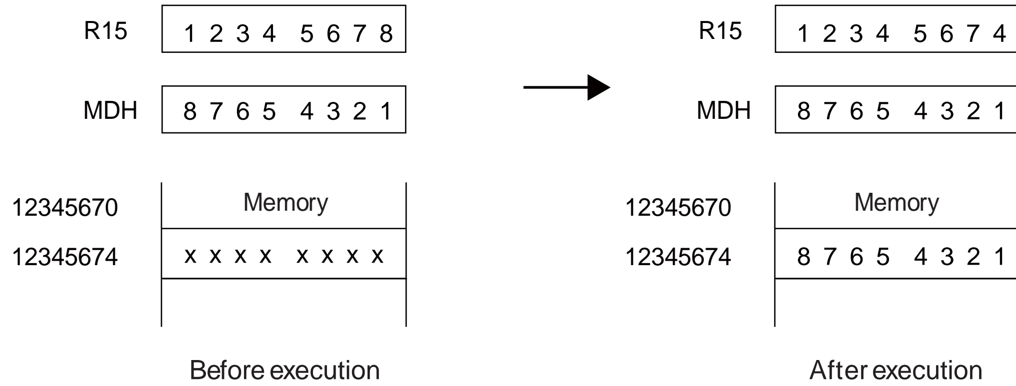
a cycle

- Instruction Format



- Execution Example

ST MDH,@-R15 ; Bit pattern of the instruction: 0001 0111 1000 0100



7.156 ST (Store Word Data in Program Status Register to Memory)

Subtracts 4 from the value of R15, stores the word data in the program status (PS) to the memory address indicated by the new value of R15.

- Assembler Format

ST PS,@-R15

- Operation

R15 - 4 → R15

PS → (R15)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

- Classification

Memory Store instruction, Instruction with delay slot

- Execution Cycles

a cycle

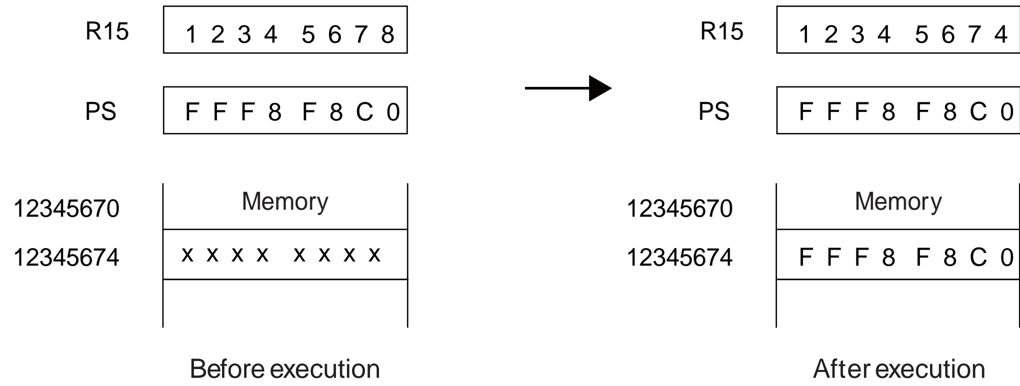
- Instruction Format

MSB LSB

0	0	0	1	0	1	1	1	1	0	0	1	0	0	0	0

● Execution Example

```
ST   PS,@-R15        ; Bit pattern of the instruction: 0001 0111 1001 0000
```



7.157 STB (Store Byte Data in Register to Memory)

Stores the byte data in Ri to memory address Rj.

● Assembler Format

STB Ri,@Rj

● Operation

Ri → (Rj)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

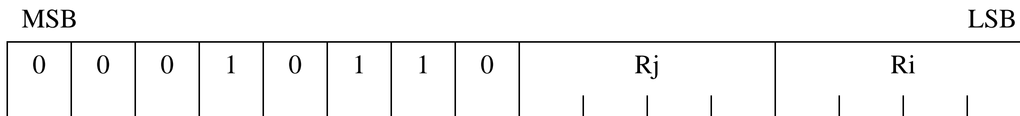
● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

a cycle

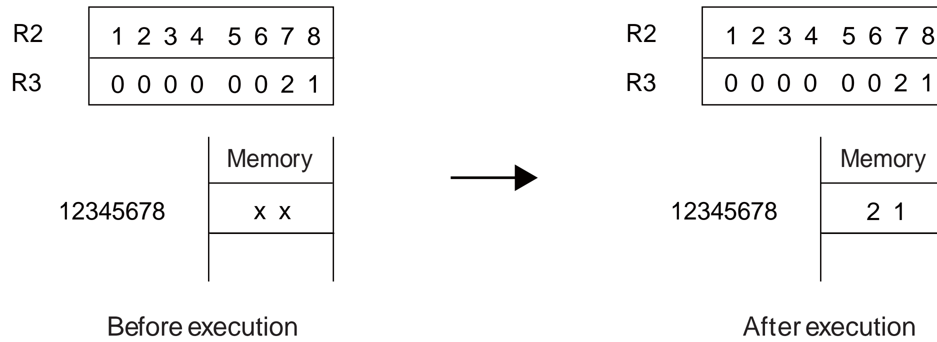
● Instruction Format



FR81 Family

● Execution Example

STB R3,@R2 ; Bit pattern of the instruction: 0001 0110 0010 0011



7.158 STB (Store Byte Data in Register to Memory)

Stores the byte data in Ri to memory address R13 + Rj.

● Assembler Format

STB Ri,@(R13, Rj)

● Operation

Ri → (R13+Rj)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

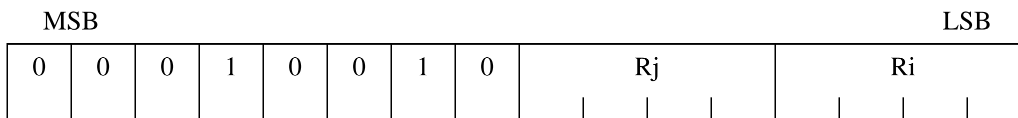
● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

a cycle

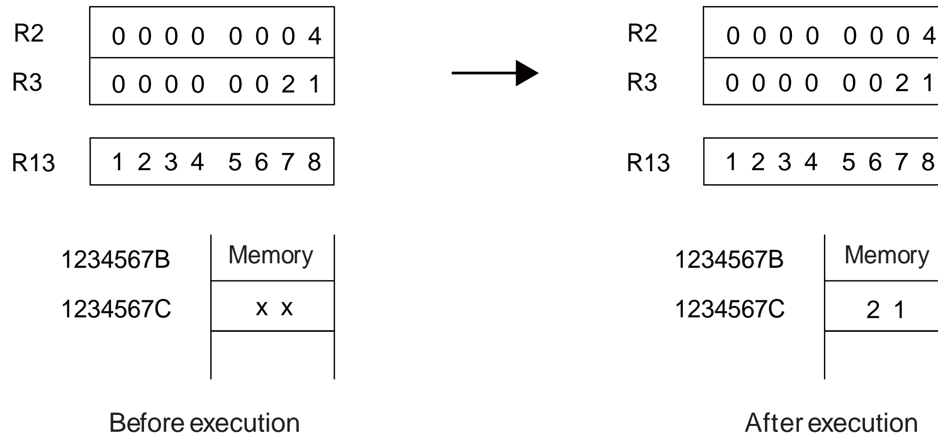
● Instruction Format



FR81 Family

● Execution Example

STB R3,@(R13, R2) ; Bit pattern of the instruction: 0001 0010 0010 0011



7.159 STB (Store Byte Data in Register to Memory)

Stores the byte data in Ri to memory address R14 + o8. The value o8 is a signed calculation. The value of o8 is specified in disp8.

● Assembler Format

STB Ri,@(R14, disp8)

● Operation

Ri → (R14+o8)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

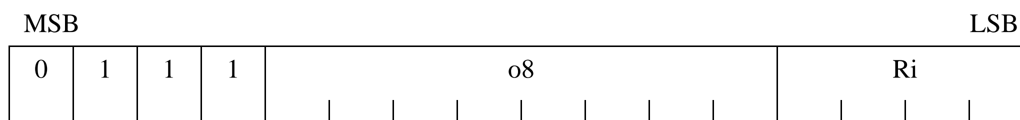
● Classification

Memory Store instruction, Instruction with delay slot

● Execution Cycles

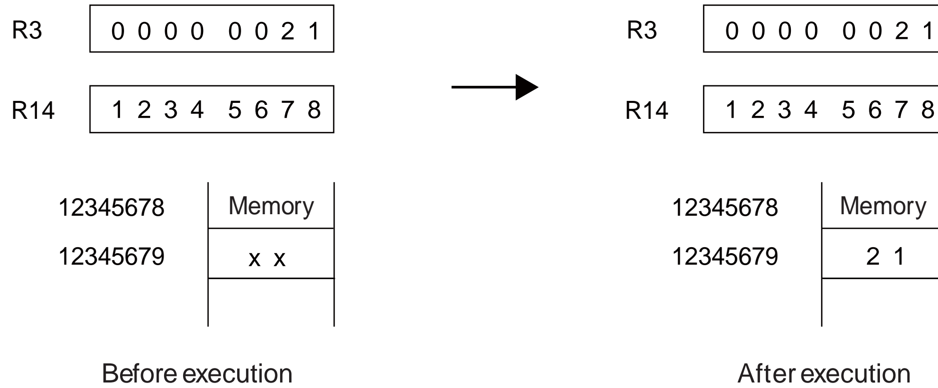
a cycle

● Instruction Format



● Execution Example

STB R3,@(R14,1) ; Bit pattern of the instruction: 0111 0000 0001 0011



7.160 STB (Store Byte Data in Register to Memory)

Loads the byte data in Ri to memory address BP+u16. Unsigned u16 value is calculated. The value in u16 is specified as udisp16.

● Assembler Format

STB Ri, @(BP, udisp16)

● Operation

Ri → (BP+u16)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

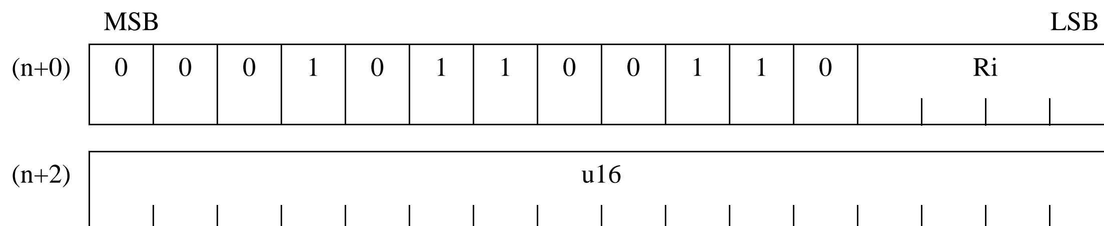
● Classification

Memory Store instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

FR81 Family

7.161 STH (Store Halfword Data in Register to Memory)

Stores the half-word data in Ri to memory address Rj.

- Assembler Format

STH Ri,@Rj

- Operation

Ri → (Rj)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

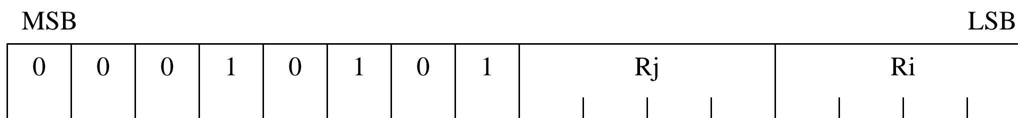
- Classification

Memory Store instruction, Instruction with delay slot

- Execution Cycles

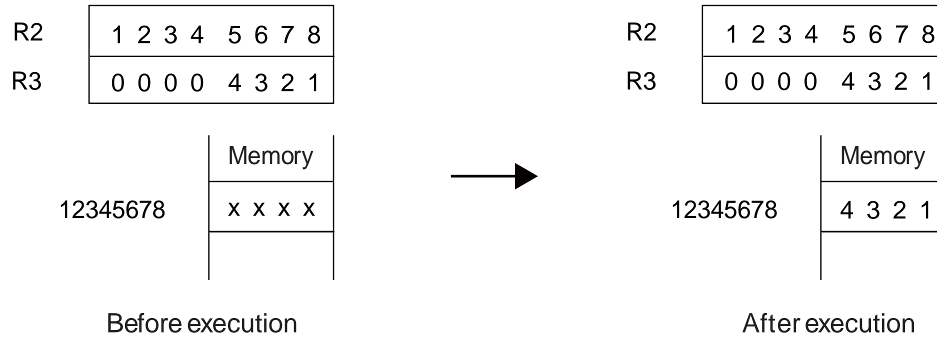
a cycle

- Instruction Format



- Execution Example

```
STH   R3,@R2        ; Bit pattern of the instruction: 0001 0101 0010 0011
```



FR81 Family

7.162 STH (Store Halfword Data in Register to Memory)

Stores the half-word data in Ri to memory address R13 + Rj.

- Assembler Format

STH Ri,@(R13, Rj)

- Operation

Ri → (R13+Rj)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

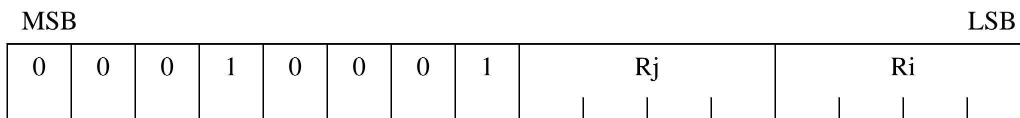
- Classification

Memory Store instruction, Instruction with delay slot

- Execution Cycles

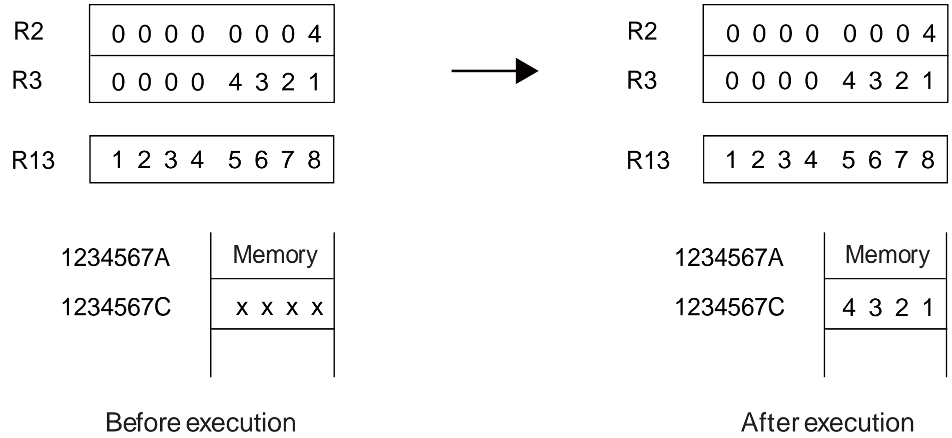
a cycle

- Instruction Format



● Execution Example

STH R3,@(R13, R2) ; Bit pattern of the instruction: 0001 0001 0010 0011



7.163 STH (Store Halfword Data in Register to Memory)

Stores the half-word data in Ri to memory address R14 + o8 × 2. The value of o8 × 2 is specified in disp9.

- Assembler Format

STH Ri,@(R14, disp9)

- Operation

Ri → (R14+o8×2)

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

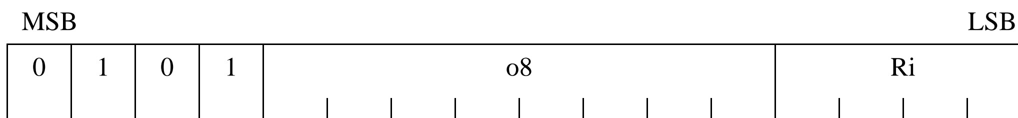
- Classification

Memory Store instruction, Instruction with delay slot

- Execution Cycles

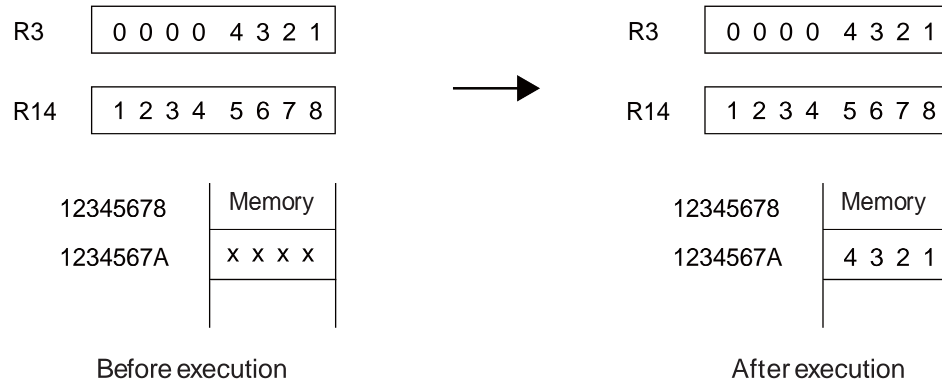
a cycle

- Instruction Format



● Execution Example

STH R3,@(R14,2) ; Bit pattern of the instruction: 0101 0000 0001 0011



FR81 Family

7.164 STH (Store Halfword Data in Register to Memory)

Loads the half word data in Ri to memory address BP+u16 × 2. Unsigned u16 value is calculated. The value in u16 × 2 is specified as udisp17.

● Assembler Format

STB Ri, @(BP, udisp17)

● Operation

Ri → (BP+u16 × 2)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

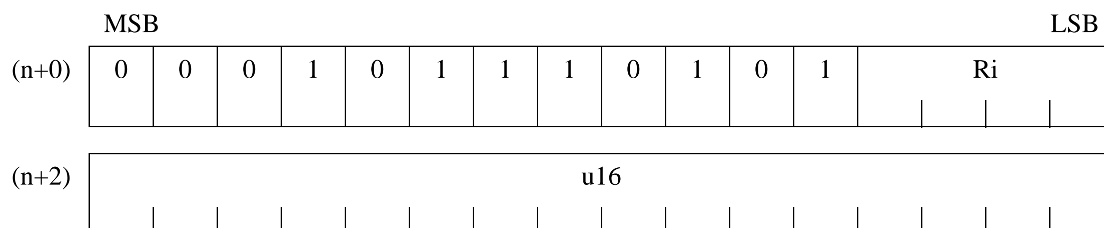
● Classification

Memory Store instruction, FR81 family

● Execution Cycles

a cycle

● Instruction Format



● EIT Occurrence and Detection

A data access protection violation exception, an invalid instruction exception (data access error), or an interrupt is detected.

7.165 STILM (Set Immediate Data to Interrupt Level Mask Register)

Transfers the immediate data to the interrupt level mask register (ILM) in the program status (PS).

- Assembler Format

STILM #u8

- Operation

```
if (ILM < 16)
    u8 → ILM
else if (u8 < 16)
    u8+16 → ILM
else
    u8 → ILM
```

This is a privilege instruction, which is only available in privilege mode. If this instruction is executed in user mode, it causes an invalid instruction exception (privilege instruction execution).

Only the lower 5 bits (bit4 to bit0) of the immediate data are valid. At the time this instruction is executed, if the value of the interrupt level mask register (ILM) is in the range 16 to 31, only new ILM settings between 16 and 31 can be entered. If the value u8 is in the range 0 to 15, the value 16 will be added to that data before being transferred to the ILM. If the original ILM value is in the range 0 to 15, then any value between 0 and 31 can be transferred to the ILM.

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

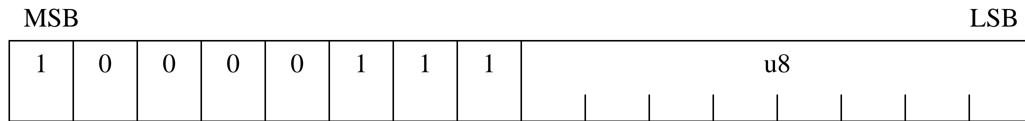
- Classification

Other instructions, Instruction with delay slot, FR81 updating

● Execution Cycles

1 cycle

● Instruction Format



● EIT Occurrence and Detection

User mode:

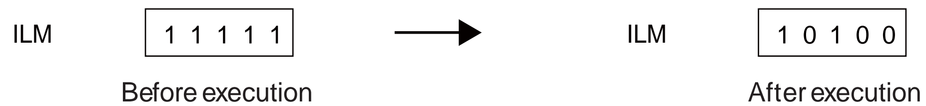
Used to generate an invalid instruction exception (privilege instruction execution).

Privilege mode:

An interrupt is detected (ILM after instruction execution is used).

● Execution Example

STILM #14H ; Bit pattern of the instruction: 1000 0111 0001 0100



7.166 STM0 (Store Multiple Registers)

The STM0 instruction stores the word data from multiple registers specified in reglist (from R0 to R7) and repeats the operation of storing the result in address R15 after subtracting the value of 4 from R15. Registers are processed in ascending order.

- Assembler Format

STM0 (reglist)

Registers from R0 to R7 are separated by "," , arranged and specified in reglist.

- Operation

The following operations are repeated according to the number of registers in reglist.

R15-4 → R15
Ri → (R15)

The bit values and register numbers for reglist (STM0) are shown in Table 7.166-1.

Table 7.166-1 Bit values and register numbers for reglist (STM0)

Bit	Register
7	R0
6	R1
5	R2
4	R3
3	R4
2	R5
1	R6
0	R7

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

Other instructions

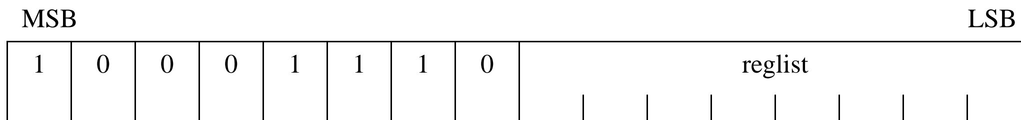
● Execution Cycles

If "n" is the number of registers specified in the parameter reglist, the execution cycles required are as follows.

When n=0: 1 cycle

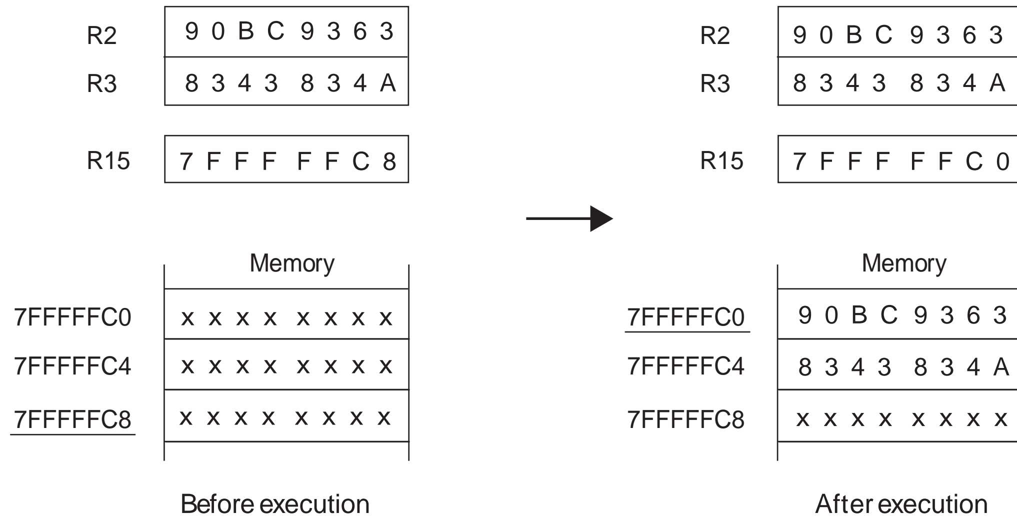
Otherwise: a × n cycles

● Instruction Format



● Execution Example

STM0 (R2, R3) ; Bit pattern of the instruction: 1000 1110 0011 0000



7.167 STM1 (Store Multiple Registers)

The STM1 instruction stores the word data from multiple registers specified in reglist (from R8 to R15) and repeats the operation of storing the result in address R15 after subtracting the value of 4 from R15. Registers are processed in ascending order. If R15 is specified in the parameter reglist, the contents of R15 retained before the instruction is executed will be written to memory.

- Assembler Format

STM1 (reglist)

Registers from R0 to R7 are separated by "," , arranged and specified.in reglist.

- Operation

The following operations are repeated according to the number of registers in reglist.

R15-4 → R15
Ri → (R15)

The bit values and register numbers for reglist (STM1) are shown in Table 7.167-1.

Table 7.167-1 Bit values and register numbers for reglist (STM1)

Bit	Register
7	R8
6	R9
5	R10
4	R11
3	R12
2	R13
1	R14
0	R15

- Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

● Classification

Other instructions

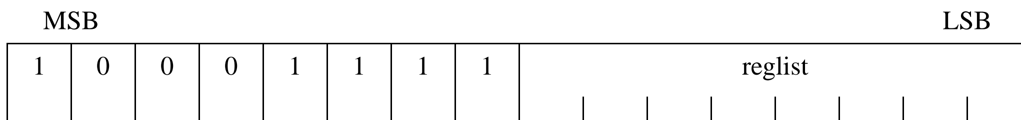
● Execution Cycles

If "n" is the number of registers specified in the parameter reglist, the execution cycles required are as follows.

When n=0: 1 cycle

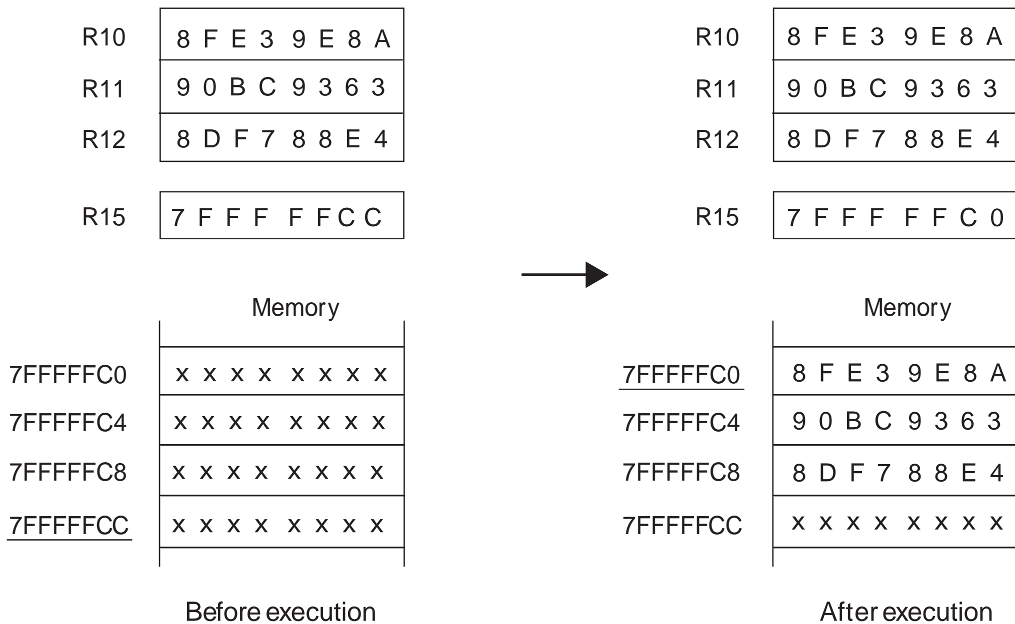
Otherwise: $a \times n$ cycles

● Instruction Format



● Execution Example

STM1 (R10, R11, R12) ; Bit pattern of the instruction: 1000 1111 0011 1000



7.168 SUB (Subtract Word Data in Source Register from Destination Register)

Subtracts the word data in Rj from the word data in Ri, stores the results to Ri.

- Assembler Format

SUB Rj, Ri

- Operation

Ri - Rj → Ri

- Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is zero, cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when a borrow has occurred as a result of the operation, cleared otherwise.

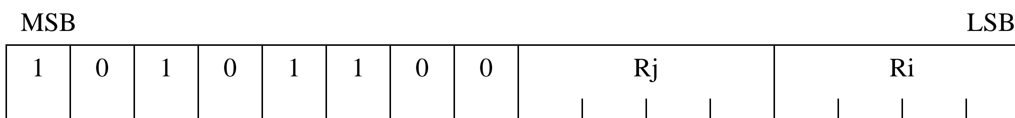
- Classification

Add/Subtract instruction, Instruction with delay slot

- Execution Cycles

1 cycle

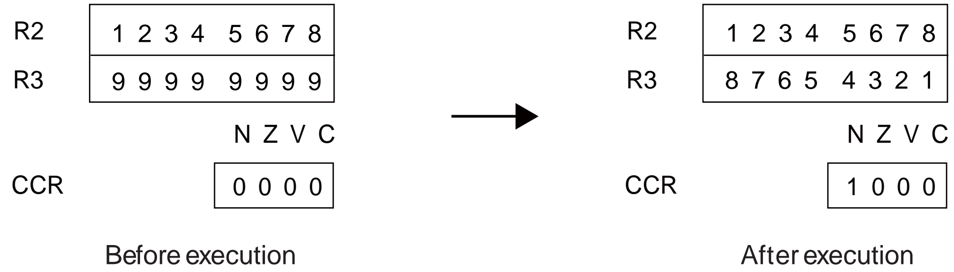
- Instruction Format



FR81 Family

● Execution Example

SUB R2, R3 ; Bit pattern of the instruction: 1010 1100 0010 0011



7.169 SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register)

Subtracts word data in Rj and carry flag (C) from Ri, stores the results to Ri.

● Assembler Format

SUBC Rj, Ri

● Operation

Ri - Rj - C → Ri

● Flag Change

N	Z	V	C
C	C	C	C

N: Set when the MSB of the operation result is "1", cleared when the MSB is "0".

Z: Set when the operation result is "0", cleared otherwise.

V: Set when an overflow has occurred as a result of the operation, cleared otherwise.

C: Set when an borrow has occurred as a result of the operation, cleared otherwise.

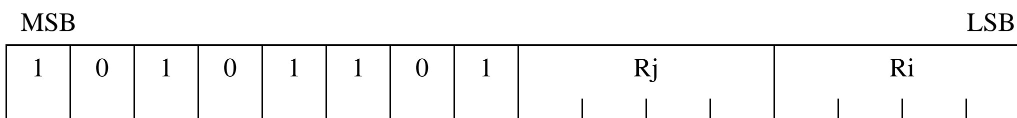
● Classification

Add/Subtract instruction, Instruction with delay slot

● Execution Cycles

1 cycle

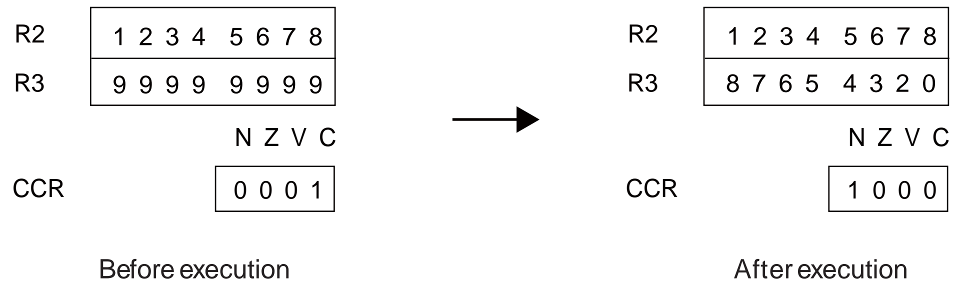
● Instruction Format



FR81 Family

- Execution Example

SUBC R2, R3 ; Bit pattern of the instruction: 1010 1101 0010 0011



7.170 SUBN (Subtract Word Data in Source Register from Destination Register)

Subtracts the word data in Rj from the word data in Ri, stores results to Ri without changing the flag settings.

● Assembler Format

SUBN Rj, Ri

● Operation

Ri - Rj → Ri

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

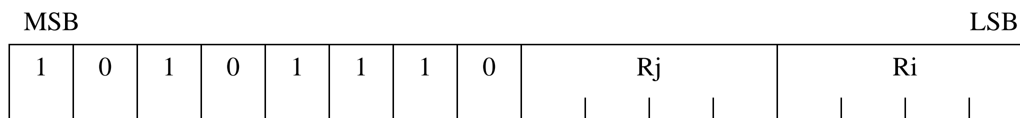
● Classification

Add/Subtract instruction, Instruction with delay slot

● Execution Cycles

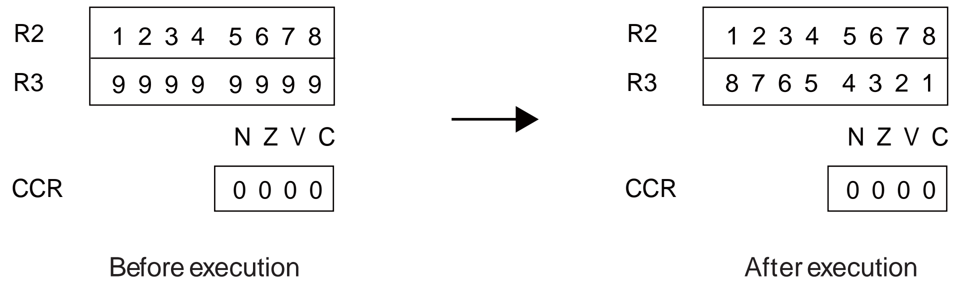
1 cycle

● Instruction Format



- Execution Example

SUBN R2, R3 ; Bit pattern of the instruction: 1010 1110 0010 0011



7.171 XCHB (Exchange Byte Data)

Exchanges the contents of the byte address indicated by Rj and those indicated by Ri. The lower 8 bits of data originally at Ri are transferred to the byte address indicated by Rj and the data originally at Rj is extended with zeros and transferred to Ri.

● Assembler Format

XCHB @Rj, Ri

● Operation

Ri → TEMP

extu((Rj)) → Ri

TEMP → (Rj)

● Flag Change

N	Z	V	C
-	-	-	-

N, Z, V, C: Unchanged.

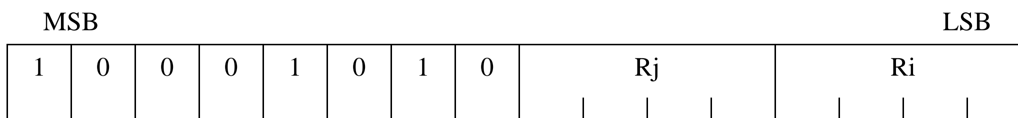
● Classification

Other instructions, Read/Modify/Write type instruction

● Execution Cycles

2a cycles

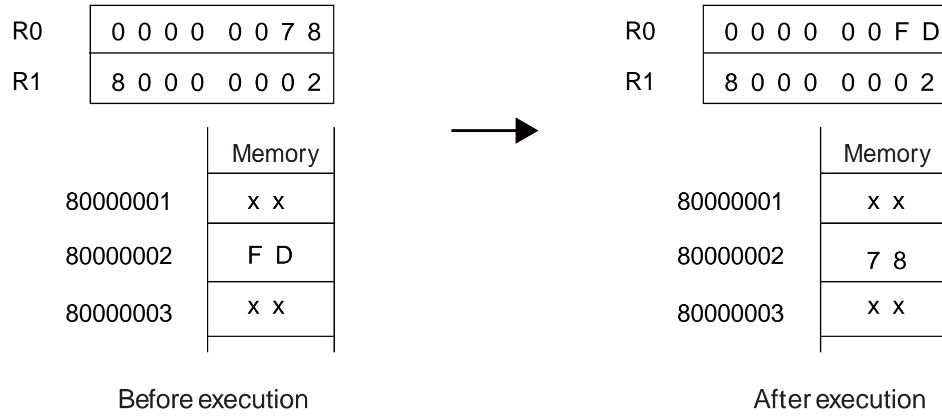
● Instruction Format



FR81 Family

- Execution Example

XCHB @R1, R0 ; Bit pattern of the instruction: 1000 1010 0001 0000



APPENDIX

It includes Instruction Lists and Instruction Maps of FR81 Family.

APPENDIX A Instruction Lists

APPENDIX B Instruction Maps

APPENDIX C Supplemental Explanation about FPU Exception Processing

APPENDIX A Instruction Lists

It includes Instruction Lists of FR81 Family CPU.

A.1 Meaning of Symbols
A.2 Instruction Lists
A.3 List of Instructions that can be positioned in the Delay Slot

A.1 Meaning of Symbols

This section describes the meaning of symbols used in the Instruction Lists and Detailed Execution Instructions has been explained.

A.1.1 Mnemonic and Operation Columns

These are the symbols used in Mnemonic and Operation columns of Instruction Lists as well as assembler format of Detailed Execution Instructions and operation.

i4

It is 4-bit immediate data. 0(0_H) to 15(F_H) in case of zero extension and -16(0_H) to -1(F_H) in case of minus extension can be specified.

Table A.1-1 zero extension and minus extension values of 4-bit immediate data

Bit Pattern	Specified Value	
	Zero Extension	Minus Extension
0000_B	0	-16
0001_B	1	-15
0010_B	2	-14
. . .		
1101_B	13	-3
1110_B	14	-2
1111_B	15	-1

i8

8-bit immediate data, Range 0 (00_H) to 255 (FF_H)

i20

20-bit immediate data, Range 0 (00000_H) to 1,048,575 ($FFFFF_H$)

i32

32-bit immediate data, Range 0 ($0000\ 0000_H$) to 4,294,967,295 ($FFFF\ FFFF_H$)

s8

signed 8-bit immediate data, range -128 (80_H) to 127 ($7F_H$)

s10

signed 10-bit immediate data, range-512 (200_H) to 508 ($1FC_H$) in multiples of 4

u4

unsigned 4-bit immediate data, range 0 (0_H) to 15 (F_H)

u8

unsigned 8-bit immediate data, range 0 (00_H) to 255 (FF_H)

u10

unsigned 10-bit immediate data, range 0 (000_H) to 1020 ($3FC_H$) in multiples of 4

udisp6

unsigned 6-bit address values, range 0 (00_H) to 60 ($3C_H$) in multiples of 4

udisp16

unsigned 16-bit address values, range 0 (0000_H) to 65535 ($FFFF_H$), range 0 (0000_H) to 65532 ($FFFC_H$) in multiples of 4

udisp17

unsigned 17-bit address values, range 0 (00000_H) to 131070 ($1FFFE_H$) in multiples of 2

udisp18

unsigned 18-bit address values, range 0 (00000_H) to 262140 ($3FFFC_H$) in multiples of 4

disp8

signed 6-bit address values, range -128(80_H) to 127($7F_H$)

disp9

signed 9-bit address values, range -256(100_H) to 254($0FE_H$) in multiples of 2

disp10

signed 10-bit address values, range -512(200_H) to 508($1FC_H$) in multiples of 4

disp16

signed 16-bit address values, range -32768 (8000_H) to 32764 ($FFFC_H$)

dir8

Unsigned 8-bit address values, range 0 (00_H) to255 (FF_H)

dir9

unsigned 9-bit address values, range 0 (000_H) to 510 ($1FE_H$) in multiples of 2

dir10

unsigned 10-bit address values, range 0 (000_H) to 1020 ($3FC_H$)in multiples of 4

label9

branch address, range 256 (100_H) to 254 ($0FE_H$) in multiples of 2 for the value of Program Counter (PC) +2

label12

branch address, range -2048 (800_H) to 2046 ($7FE_H$) in multiples of 2 for the value of Program Counter (PC) +2

label17

branch address, range -65536 (10000_H) to 65534 ($0FFFE_H$) in multiples of 2 for the value of Program Counter (PC) +2

label21

branch address, range -1048576 (100000_H) to 1048574($0FFFFE_H$) in multiples of 2 for the value of Program Counter (PC) +2

rel8

signed 8-bit relative address. Result which is double the value of rel8 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range 128 (80_H) to 127 ($7F_H$)

rel11

signed 11-bit relative address. Result which is double the value of rel11 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range -1024 (400_H) to 1023 ($3FF_H$)

rel16

signed 16-bit relative address. Result which is double the value of rel16 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range -32768 (8000_H) to 32767 ($7FFF_H$)

rel20

signed 20-bit relative address. Result which is double the value of rel20 for the value of Program Counter (PC) +2 will denote the Branch Destination Address. Range -524288 (80000_H) to 524287 ($7FFFF_H$)

Ri, Rj

Indicates General-purpose Registers (R0 to R15)

Table A.1-2 Specification of General-purpose register based on Rj/Ri

Ri / Rj	Register	Ri / Rj	Register
0000	R0	1000	R8
0001	R1	1001	R9
0010	R2	1010	R10
0011	R3	1011	R11
0100	R4	1100	R12
0101	R5	1101	R13
0110	R6	1110	R14
0111	R7	1111	R15

Rs

Indicates Dedicated Registers (TBR, RP, USP, SSP, MDH, MDL, BP, FCR, ESR, DBR)

Table A.1-3 Specification of Dedicated Register based on Rs

Rs	Register	Rs	Register
0000	Table Base Register (TBR)	1000	Exception status register (ESR)
0001	Return Pointer (RP)	1001	Reserved (Disabled)
0010	System Stack Pointer (SSP)	1010	
0011	User Stack Pointer (USP)	1011	
0100	Multiplication/Division Register (MDH)	1100	
0101	Multiplication/Division Register (MDL)	1101	
0110	Base pointer (BP)	1110	
0111	FPU control register (FCR)	1111	Debug register (DBR)

FRi, FRj, FRk

Indicates Floating Point Registers (FR0 to FR15)

Table A.1-4 Floating Point Register based on FRi/FRj/FRk

FRi/FRj/FRk	Register
0000	FR0
0001	FR1
0010	FR2
0011	FR3
0100	FR4
0101	FR5
0110	FR6
0111	FR7

FRi/FRj/FRk	Register
1000	FR8
1001	FR9
1010	FR10
1011	FR11
1100	FR12
1101	FR13
1110	FR14
1111	FR15

(reglist)

Indicates 8-bit Register list. General purpose register corresponding to each bit value can be specified.

Table A.1-5 Correspondence between reglist of LDM0,LDM1 Instruction and General purpose Register

LDM0 Instruction	
reglist	Register
bit0	R0
bit1	R1
bit2	R2
bit3	R3
bit4	R4
bit5	R5
bit6	R6
bit7	R7

LDM1Instruction	
reglist	Register
bit0	R8
bit1	R9
bit2	R10
bit3	R11
bit4	R12
bit5	R13
bit6	R14
bit7	R15

Table A.1-6 Correspondence between reglist of STM0,STM1 Instruction and General purpose Register

STM0 Instruction	
reglist	Register
bit0	R7
bit1	R6
bit2	R5
bit3	R4
bit4	R3
bit5	R2
bit6	R1
bit7	R0

STM1 Instruction	
reglist	Register
bit0	R15
bit1	R14
bit2	R13
bit3	R12
bit4	R11
bit5	R10
bit6	R9
bit7	R8

(frlist)

Indicates 16-bit Register list. Floating Point Registers (FR0 to FR15) corresponding to each bit value can be specified.

Table A.1-7 Correspondence between frlist bit of FLDM Instruction and Floating Point Registers

frlist	Register	frlist	Register
bit0	FR0	bit8	FR8
bit1	FR1	bit9	FR9
bit2	FR2	bit10	FR10
bit3	FR3	bit11	FR11
bit4	FR4	bit12	FR12
bit5	FR5	bit13	FR13
bit6	FR6	bit14	FR14
bit7	FR7	bit15	FR15

Table A.1-8 Correspondence between frlist bit of FSTM Instruction and Floating Point Registers

frlist	Register	frlist	Register
bit0	FR15	bit8	FR7
bit1	FR14	bit9	FR6
bit2	FR13	bit10	FR5
bit3	FR12	bit11	FR4
bit4	FR11	bit12	FR3
bit5	FR10	bit13	FR2
bit6	FR9	bit14	FR1
bit7	FR8	bit15	FR0

A.1.2 Operation Column

These are symbols used in Operation Column of Instruction Lists and operation of Detailed Execution Instructions.

extu( )

indicates a zero extension operation, in which portion lacking higher bits is complimented by adding "0" bit.

extn( )

indicates a minus extension operation, in which portion lacking higher bits is complimented by adding "1" bit.

exts( )

indicates a sign extension operation, in which zero extension is performed for the data within ( ) if MSB

is "0" and a minus extension is performed if MSB is "1".

&

Indicates logical calculation of each bit (AND)

|

Indicates the logical sum of each bit (OR)

^

Indicates Dedicated Logical Sum of each bit (EXOR)

( )

Indicates specification of indirect address. It is address memory read/write value of the Register or formula within ( ).

{ }

Indicate the calculation priority. Since ( ) is used for specifying indirect address, different bracket namely { } is used.

if (Condition) then {formula} or if (condition) then {Formula 1} else {Formula 2}

Indicates the execution of conditions. If the conditions are established, formula after ‘then’ is executed and when the conditions are not established, formula next to ‘else’ is executed. Formula can be described variously using the { }.

[m:n]

Bits from m to n are extracted and targeted for operation.

A.1.3 Format Column

Symbols used in the Format Column of the Instruction Lists.

A to N

Indicates the Instruction Formats. A to N correspond to TYPE-A to TYPE-N.

A.1.4 OP Column

Hexadecimal value used in the Instruction Lists. They denote operation codes (OP). They branch into the following depending on the Instruction Format.

TYPE-A, TYPE-C, TYPE-D, TYPE-G

2-digit hexadecimal value represents 8-bit OP code

TYPE-B

2-digit hexadecimal value represents 4 bits of OP code as higher 1 digit and "0" for lower digit.

TYPE-E, TYPE-H, TYPE-I, TYPE-J, TYPE-K

3-digit hexadecimal value represents 12-bit OP code.

TYPE-F

2-digit hexadecimal value represents 8 bits with 3-bit 000_B added below 5-bit OP code.

TYPE-L, TYPE-N

4-digit hexadecimal value represents 16 bits with 2-bit 00_B added below 14-bit OP code.

TYPE-M

4-digit hexadecimal value represents 16-bit OP code.

A.1.5 CYC Column

Symbols used in CYC Column of Instruction Lists and execution cycles of Detailed Execution Instructions. Numerical values represent CPU clock cycles. Minimum of a to d is 1 cycle.

a

Memory access cycles. Cycles change depending on the access target. Minimum value is 1 cycle.

b

Memory access cycles. Cycles change depending on the access target. Minimum value is 1 cycle.

It is 1 cycle when uncompleted LD Instructions are less than 4 Instructions and Register which is the object of load operation is not referred by the subsequent Instruction.

When uncompleted LD Instructions become more than 4 in number, an interlock will be applied from that point till the completion of first LD Instruction and the number of execution cycles will be increased by (Memory Access Cycles - Number of cycles from the issue of an Instruction till first LD Instruction is completed).

When the Register which is target of load operation is referred to by the succeeding Instruction, an interlock will be applied from that point and the number of execution cycles will increase by (Memory Access Cycles - Number of cycle from the issue of an Instruction till an instruction refers to the targeted Register + 1).

c

An interlock will be applied when the immediately next Instruction refers to Multiplication/Division Register (MDH) and the number of execution cycles will be increased to 2. Otherwise it will be 1 cycle.

d

There will be 2 cycles when pre-fetching of Instruction in the Pre-fetch Buffer is not carried out. Minimum value is 1 cycle.

A.1.6 FLAG Column

Symbols used for flag change in the Flag Column of Instruction Lists and Detailed Execution Instructions. Represents change in Negative Flag (N), Zero Flag (Z), Overflow Flag (V), Carry Flag (C) of the Condition Code Register (CCR).

C

Varies depending on the result of operation

-

No change

0

Value becomes "0"

1

Value becomes "1"

A.1.7 RMW Column

Symbols used in the RMW Column of Instruction Lists. It represents whether or not it is Read-Modify-Write Instruction.

-

Instruction is not Read-Modify-Write Instruction.

❍

Instruction is Read-Modify-Write Instruction.

A.1.8 Reference Column

Represents the portion explained in “Chapter 7 Detailed Execution Instructions”

A.2 Instruction Lists

This part indicates Instruction Lists of FR81 Family CPU.

There are a total of 231 instructions in FR81 Family CPU. These instructions are divided into the following 21 categories.

- Add/Subtract Instructions (10Instructions)
- Compare Calculation Instructions (3 Instructions)
- Logical Calculation Instructions (12 Instructions)
- Bit Operation Instructions (8 Instructions)
- Multiply/ Divide Instructions (10 Instructions)
- Shift Instructions (9 Instructions)
- Immediate Data Transfer Instructions (3 Instructions)
- Memory Load Instructions (16 Instructions)
- Memory Store Instructions (16 Instructions)
- Inter-Register Transfer Instructions/Dedicated Register Transfer Instructions (5 Instructions)
- Non-delayed Branching Instructions (24 Instructions)
- Delayed Branching Instructions (21 Instructions)
- Direct Addressing Instructions (14 Instructions)
- Bit Search Instructions (3 Instructions)
- Other Instructions (16 Instructions)
- FPU Memory Load Instructions (7 Instructions)
- FPU Memory Store Instructions (7 Instructions)
- FPU Single-Precision Floating Point Calculation Instruction (12 Instructions)
- FPU Inter-Register Transfer Instruction (3 Instructions)
- FPU Branching Instruction without Delay (16 Instructions)
- FPU Branching Instruction with Delay (16 Instructions)

Table A.2-1 Add/Subtract Instructions (10Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
ADD Rj, Ri	A	A6	1	CCCC	-	Ri+Rj → Ri		7.2
ADD #i4, Ri	C	A4	1	CCCC	-	Ri+extu(i4) → Ri	i4 is zero extension	7.1
ADD2 #i4, Ri	C	A5	1	CCCC	-	Ri+extn(i4) → Ri	i4 is Minus extension	7.3
ADDC Rj, Ri	A	A7	1	CCCC	-	Ri+Rj+C → Ri	Add with carry	7.4
ADDN Rj, Ri	A	A2	1	----	-	Ri+Rj → Ri		7.6
ADDN #i4, Ri	C	A0	1	----	-	Ri+extu(i4) → Ri	i4 is Zero extension	7.5
ADDN2 #i4, Ri	C	A1	1	----	-	Ri+extn(i4) → Ri	i4 is Minus extension	7.7
SUB Rj, Ri	A	AC	1	CCCC	-	Ri-Rj → Ri		7.129
SUBC Rj, Ri	A	AD	1	CCCC	-	Ri-Rj-C →Ri	Add with carry	7.130
SUBN Rj, Ri	A	AE	1	----	-	Ri-Rj → Ri		7.131

Table A.2-2 Compare Calculation Instructions (3 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
CMP Rj, Ri	A	AA	1	CCCC	-	Ri-Rj		7.32
CMP #i4, Ri	C	A8	1	CCCC	-	Ri-extu(i4)	i4 is Zero extension	7.31
CMP2 #i4, Ri	C	A9	1	CCCC	-	Ri-extn(i4)	i4 is Minus extension	7.33

Table A.2-3 Logical Calculation Instructions (12 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
AND Rj, Ri	A	82	1	CC--	-	Ri & Rj → Ri	Word	7.10
AND Rj, @Ri	A	84	1+2a	CC--	❍	(Ri) & Rj → (Ri)	Word	7.9
ANDH Rj, @Ri	A	85	1+2a	CC--	❍	(Ri) & Rj → (Ri)	Half-Word	7.13
ANDB Rj, @Ri	A	86	1+2a	CC--	❍	(Ri) & Rj → (Ri)	Byte	7.11
OR Rj, Ri	A	92	1	CC--	-	Ri	Rj → Ri	Word	7.139
OR Rj, @Ri	A	94	1+2a	CC--	❍	(Ri)	Rj → (Ri)	Word	7.138
ORH Rj, @Ri	A	95	1+2a	CC--	❍	(Ri)	Rj → (Ri)	Half-Word	7.142
ORB Rj, @Ri	A	96	1+2a	CC--	❍	(Ri)	Rj → (Ri)	Byte	7.140
EOR Rj, Ri	A	9A	1	CC--	-	Ri ^ Rj → Ri	Word	7.56
EOR Rj, @Ri	A	9C	1+2a	CC--	❍	(Ri) ^ Rj → (Ri)	Word	7.55
EORH Rj, @Ri	A	9D	1+2a	CC--	❍	(Ri) ^ Rj → (Ri)	Half-Word	7.58
EORB Rj, @Ri	A	9E	1+2a	CC--	❍	(Ri) ^ Rj → (Ri)	Byte	7.57

Table A.2-4 Bit Operation Instructions (8 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
BANDL #u4, @Ri	C	80	1+2a	----	❍	(Ri) & {$F0_H$+u4} → (Ri)	Lower 4- bit	7.18
BANDH #u4, @Ri	C	81	1+2a	----	❍	(Ri) & {u4<<4+$0F_H$} → (Ri)	Higher 4 bit	7.17
BORL #u4, @Ri	C	90	1+2a	----	❍	(Ri) | u4 → (Ri)	Lower 4- bit	7.24
BORH #u4, @Ri	C	91	1+2a	----	❍	(Ri) | {u4<<4} → (Ri)	Higher 4 bit	7.23
BEORL #u4, @Ri	C	98	1+2a	----	❍	(Ri) ^ u4 → (Ri)	Lower 4- bit	7.22
BEORH #u4, @Ri	C	99	1+2a	----	❍	(Ri) ^ {u4<<4} → (Ri)	Higher 4 bit	7.21
BTSTL #u4, @Ri	C	88	2+a	0C--	-	(Ri) & u4	Lower 4- bit	7.26
BTSTH #u4, @Ri	C	89	2+a	CC--	-	(Ri) & {u4<<4}	Higher 4 bit	7.25

Table A.2-5 Multiply/ Divide Instructions (10 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
MUL Rj, Ri	A	AF	5	CCC-	-	Ri × Rj → MDH,MDL	32 × 32 bit = 64 bit	7.133
MULU Rj, Ri	A	AB	5	CCC-	-	Ri × Rj → MDH,MDL	Unsigned	7.135
MULH Rj, Ri	A	BF	3	CC--	-	Ri × Rj → MDL	16 × 16 bit = 32 bit	7.134
MULUH Rj, Ri	A	BB	3	CC--	-	Ri × Rj → MDL	Unsigned	7.136
DIV0S Ri	E	97-4	1	----	-	In the Specified Instruction Sequence MDL ÷ Ri → MDL MDL%Ri → MDH	Step Calculation 32 ÷ 32 bit = 32 bit	7.34
DIV0U Ri	E	97-5	1	----	-			7.35
DIV1 Ri	E	97-6	1	-C-C	-			7.36
DIV2 Ri	E	97-7	c	-C-C	-			7.37
DIV3	E’	9F-6	1	----	-			7.38
DIV4S	E’	9F-7	1	----	-			7.39

Table A.2-6 Shift Instructions (9 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
LSL Rj, Ri	A	B6	1	CC-C	-	Ri << Rj → Ri	Logical Shift	7.120
LSL #u4, Ri	C	B4	1	CC-C	-	Ri << u4 → Ri		7.121
LSL2 #u4, Ri	C	B5	1	CC-C	-	Ri << {u4+16} → Ri		7.122
LSR Rj, Ri	A	B2	1	CC-C	-	Ri >> Rj → Ri	Logical Shift	7.123
LSR #u4, Ri	C	B0	1	CC-C	-	Ri >> u4 → Ri		7.124
LSR2 #u4, Ri	C	B1	1	CC-C	-	Ri >> {u4+16} → Ri		7.125
ASR Rj, Ri	A	BA	1	CC-C	-	Ri >> Rj → Ri	Arithmetic Shift	7.14
ASR #u4, Ri	C	B8	1	CC-C	-	Ri >> u4 → Ri		7.15
ASR2 #u4, Ri	C	B9	1	CC-C	-	Ri >> {u4+16} → Ri		7.16

Table A.2-7 Immediate Data Transfer Instructions (3 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
LDI:32 #i32, Ri	H	9F-8	d	----	-	i32 → Ri		7.107
LDI:20 #i20, Ri	G	9B	d	----	-	extu(i20) → Ri	Higher 12-Bits are Zero extension	7.106
LDI:8 #i8, Ri	B	C0	1	----	-	extu(i8) → Ri	Higher 24-Bits are Zero extension	7.108

Table A.2-8 Memory Load Instructions (16 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
LD @Rj, Ri	A	04	b	----	-	(Rj) → Ri	Word	7.98
LD @(R13, Rj), Ri	A	00	b	----	-	(R13+Rj) → Ri		7.99
LD @(R14, disp10), Ri	B	20	b	----	-	(R14+o8 × 4) → Ri		7.100
LD @(R15, udisp6), Ri	C	03	b	----	-	(R15+u4 × 4) → Ri		7.101
LD @R15+, Ri	E	07-0	b	----	-	(R15) → Ri, R15+4 → R15		7.102
LD @R15+, Rs	E	07-8	b	----	-	(R15) → Rs, R15+4 → R15		7.104
LD @R15+, PS	E	07-9	1+a	CCCC	-	(R15) → PS, R15+4 → R15		7.105
LD @(BP, udisp18), Ri	J	07-4	a	----	-	(BP+u16 × 4) → Ri		7.103
LDUH @Rj, Ri	A	05	b	----	-	extu((Rj)) → Ri	Half-Word Zero extension	7.115
LDUH @(R13, Rj), Ri	A	01	b	----	-	extu((R13+Rj)) → Ri		7.116
LDUH @(R14, disp9), Ri	B	40	b	----	-	extu((R14+o8 × 2)) → Rj		7.117
LDUH @(BP, udisp17), Ri	J	07-5	a	----	-	(BP+u16 × 2) → Ri		7.118
LDUB @Rj, Ri	A	06	b	----	-	extu((Rj)) → Ri	Byte Zero extension	7.111
LDUB @(R13, Rj), Ri	A	02	b	----	-	extu((R13+Rj)) → Ri		7.112
LDUB @(R14, disp8), Ri	B	60	b	----	-	extu((R14+o8)) → Ri		7.113
LDUB @(BP, udisp16), Ri	J	07-6	a	----	-	(BP+u16) → Ri		7.114

- Relation of field o8 in the Instruction Format TYPE-B to the values disp8 to disp10 in assembly notation is as follows.

  o8 = disp8

  o8 = disp9 >> 1

  o8 = disp10 >> 2

- Relation of field u4 in the Instruction Format TYPE-C to the values udisp6 in assembly notation is as follows.

  u4 = udisp6 >> 2

- Relation of field u16 in the Instruction Format TYPE-J to the values udisp16 to udisp18 in assembly notation is as follows.

  u16 = udisp16

  u16 = udisp17 >> 1

  u16 = udisp18 >> 2

Table A.2-9 Memory Store Instructions (16 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
ST Ri, @Rj	A	14	a	----	-	Ri → (Rj)	Word	7.149
ST Ri, @(R13, Rj)	A	10	a	----	-	Ri → (R13+Rj)		7.150
ST Ri, @(R14, disp10)	B	30	a	----	-	Ri → (R14+o8 × 4)		7.151
ST Ri, @(R15, udisp6)	C	13	a	----	-	Ri → (R15+u4 × 4)		7.152
ST Ri, @-R15	E	17-0	a	----	-	R15-4 → R15, Ri → (R15)		7.153
ST Rs, @-R15	E	17-8	a	----	-	R15-4 → R15, Rs → (R15)		7.155
ST PS, @-R15	E	17-9	a	----	-	R15-4 → R15, PS → (R15)		7.156
ST Ri, @(BP, udisp18)	J	17-4	a	----	-	Ri → (BP+u16 × 4)		7.154
STH Ri, @Rj	A	15	a	----	-	Ri → (Rj)	Half-Word	7.161
STH Ri, @(R13, Rj)	A	11	a	----	-	Ri → (R13+Rj)		7.162
STH Ri, @(R14, disp9)	B	50	a	----	-	Ri → (R14+o8 × 2)		7.163
STH Ri, @(BP, udisp17)	J	17-5	a	----	-	Ri → (BP+u16 × 2)		7.164
STB Ri, @Rj	A	16	a	----	-	Ri → (Rj)	Byte	7.157
STB Ri, @(R13, Rj)	A	12	a	----	-	Ri → (R13+Rj)		7.158
STB Ri, @(R14, disp8)	B	70	a	----	-	Ri → (R14+o8)		7.159
STB Ri, @(BP, udisp16)	J	17-6	a	----	-	Ri → (BP+u16)		7.160

- Relation of field o8 in the Instruction Format TYPE-B to the values disp8 to disp10 in assembly notation is as follows.

  o8 = disp8

  o8 = disp9 >> 1

  o8 = disp10 >> 2

- Relation of field u4 in the Instruction Format TYPE-C to the values udisp6 in assembly notation is as follows.

  u4 = udisp6 >> 2

- Relation of field u16 in the Instruction Format TYPE-J to the values udisp16 to udisp18 in assembly notation is as follows.

  u16 = udisp16

  u16 = udisp17 >> 1

  u16 = udisp18 >> 2

Table A.2-10 Inter-Register Transfer Instructions/Dedicated Register Transfer Instructions (5 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
MOV Rj, Ri	A	8B	1	----	-	Rj → Ri	Transfer between general-purpose Registers	7.126
MOV Rs, Ri	A	B7	1	----	-	Rs → Ri	Rs: Dedicated Register	7.127
MOV Ri, Rs	A	B3	1	----	-	Ri → Rs	Rs: Dedicated Register	7.129
MOV PS, Ri	E	17-1	1	----	-	PS → Ri	PS: Program Status	7.128
MOV Ri, PS	E	07-1	c	CCCC	-	Ri → PS	PS: Program Status	7.130

Table A.2-11 Non-delayed Branching Instructions (24 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
JMP @Ri	E	97-0	2	----	-	Ri → PC		7.94
CALL label12	F	D0	2	----	-	PC+2 → RP, PC+2+exts(rel11 × 2) → PC		7.27
CALL @Ri	E	97-1	2	----	-	PC+2 → RP, Ri → PC		7.28
LCALL label21	I	07-2	2	----	-	PC+4 → RP PC+4+exts(rel20 × 2) → PC		7.96
RET	E’	97-2	2	----	-	RP → PC		7.143
INT #u8	D	1F	1+3a	----	-	SSP-4 → SSP, PS → (SSP), SSP-4 → SSP, PC+2 → (SSP), 0 → CCR:I, 0 → CCR:S, (TBR+3FC-u8 × 4) → PC		7.92
INTE	E’	9F-3	1+3a	----	-	SSP → SSP, PS → (SSP), SSP → SSP, PC+2 → (SSP), 0 → CCR:S, 4 → ILM, (TBR+3D8) → PC		7.93
RETI	E’	97-3	1+2b	----	-	(SSP) → PC, SSP+4 → SSP, (SSP) → PS, SSP+4 → SSP		7.145
BNO label9	D	E1	1	----	-	No branch		7.19
BRA label9	D	E0	2	----	-	PC+2+exts(rel8 × 2) → PC		7.19
BEQ label9	D	E2	2/1	----	-	if (Z==1) then PC+2+exts(rel8 × 2) → PC		7.19
BNE label9	D	E3	2/1	----	-	if (Z==0) then PC+2+exts(rel8 × 2) → PC		7.19
BC label9	D	E4	2/1	----	-	if (C==1) then PC+2+exts(rel8 × 2) → PC		7.19
BNC label9	D	E5	2/1	----	-	if (C==0) then PC+2+exts(rel8 × 2) → PC		7.19
BN label9	D	E6	2/1	----	-	if (N==1) then PC+2+exts(rel8 × 2) → PC		7.19
BP label9	D	E7	2/1	----	-	if (N==0) then PC+2+exts(rel8 × 2) → PC		7.19
BV label9	D	E8	2/1	----	-	if (V==1) then PC+2+exts(rel8 × 2) → PC		7.19
BNV label9	D	E9	2/1	----	-	if (V==0) then PC+2+exts(rel8 × 2) → PC		7.19
BLT label9	D	EA	2/1	----	-	if (V ^ N==1) then PC+2+exts(rel8 × 2) → PC		7.19
BGE label9	D	EB	2/1	----	-	if (V ^ N==0) then PC+2+exts(rel8 × 2) → PC		7.19
BLE label9	D	EC	2/1	----	-	if ({V ^ N} | Z==1) then PC+2+exts(rel8 × 2) → PC		7.19
BGT label9	D	ED	2/1	----	-	if ({V ^ N} | Z==0) then PC+2+exts(rel8 × 2) → PC		7.19
BLS label9	D	EE	2/1	----	-	if (C or Z==1) then PC+2+exts(rel8 × 2) → PC		7.19
BHI label9	D	EF	2/1	----	-	if (C or Z==0) then PC+2+exts(rel8 × 2) → PC		7.19

- The value of "2/1" in CYC Column indicates 2 cycles if branching and 1 if not branching.
- It is necessary to set the Stack Flag (S) to "0" for RETI instruction execution.

- The field rel8 in TYPE-D Instruction Format and the field rel11 in TYPE-F Format have the following relation to the values of label9, label12 in assembly notation.
  rel8 = (label9-PC-2)/2
  rel11 = (label12-PC-2)/2
- The field rel20 in TYPE-I Instruction Format has the following relation to the values of label21 in assembly notation.
  rel20 = (labe21-PC-4)/2

Table A.2-12 Delayed Branching Instructions (21 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
JMP:D @Ri	E	9F-0	1	----	-	Ri → PC		7.95
CALL:D label12	F	D8	1	----	-	PC+4 → RP, PC+2+exts(rel11 × 2) → PC		7.29
CALL:D @Ri	E	9F-1	1	----	-	PC+4 → RP, Ri → PC		7.30
LCALL:D label21	I	17-2	1	----	-	PC+6 → RP PC+4+exts(rel20 × 2) → PC		7.97
RET:D	E’	9F-2	1	----	-	RP → PC		7.144
BNO:D label9	D	F1	1	----	-	No branch		7.20
BRA:D label9	D	F0	1	----	-	PC+2+exts(rel8 × 2) → PC		7.20
BEQ:D label9	D	F2	1	----	-	if (Z==1) then PC+2+exts(rel8 × 2) → PC		7.20
BNE:D label9	D	F3	1	----	-	if (Z==0) then PC+2+exts(rel8 × 2) → PC		7.20
BC:D label9	D	F4	1	----	-	if (C==1) then PC+2+exts(rel8 × 2) → PC		7.20
BNC:D label9	D	F5	1	----	-	if (C==0) then PC+2+exts(rel8 ×) → PC		7.20
BN:D label9	D	F6	1	----	-	if (N==1) then PC+2+exts(rel8 × 2) → PC		7.20
BP:D label9	D	F7	1	----	-	if (N==0) then PC+2+exts(rel8 × 2) → PC		7.20
BV:D label9	D	F8	1	----	-	if (V==1) then PC+2+exts(rel8 × 2) → PC		7.20
BNV:D label9	D	F9	1	----	-	if (V==0) then PC+2+exts(rel8 × 2) → PC		7.20
BLT:D label9	D	FA	1	----	-	if (V ^ N==1) then PC+2+exts(rel8 × 2) → PC		7.20
BGE:D label9	D	FB	1	----	-	if (V ^ N==0) then PC+2+exts(rel8 × 2) → PC		7.20
BLE:D label9	D	FC	1	----	-	if ({V ^ N} | Z==1) then PC+2+exts(rel × 2) → PC		7.20
BGT:D label9	D	FD	1	----	-	if ({V ^ N} | Z==0) then PC+2+exts(rel8 × 2) → PC		7.20
BLS:D label9	D	FE	1	----	-	if (C or Z==1) then PC+2+exts(rel8 × 2) → PC		7.20
BHI:D label9	D	FF	1	----	-	if (C or Z==0) then PC+2+exts(rel8 × 2) → PC		7.20

- Delayed Branching Instructions are branched after always executing the following Instruction (the Delay Slot).
- The field rel8 in TYPE-D instruction format and the field rel11 in TYPE-D format have the following relation to the values label9, label12 in assembly notation.

  rel8 = (label9-PC-2)/2

  rel11 = (label12-PC-2)/2
- The field rel20 in TYPE-I Instruction Format has the following relation to the values of label21 in assembly notation.

  rel20 = (labe21-PC-4)/2

Table A.2-13 Direct Addressing Instructions (14 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
DMOV @dir10, R13	D	08	b	----	-	(dir8 × 4) → R13	Word	7.40
DMOV R13, @dir10	D	18	a	----	-	R13 → (dir8 × 4)		7.41
DMOV @dir10, @R13+	D	0C	1+2a	----	-	(dir8 × 4) → (R13), R13+4 → (R13)		7.42
DMOV @R13+, @dir10	D	1C	1+2a	----	-	(R13) → (dir8 × 4), R13+4 → (R13)		7.43
DMOV @dir10, @-R15	D	0B	1+2a	----	-	R15-4 → (R15), (dir8 × 4) → (R15)		7.44
DMOV @R15+, @dir10	D	1B	1+2a	----	-	(R15) → (dir8 × 4), R15+4 → (R15)		7.45
DMOVH @dir9, R13	D	09	b	----	-	(dir8 × 2) → R13	Half-Word	7.50
DMOVH R13, @dir9	D	19	a	----	-	R13 → (dir8 × 2)		7.51
DMOVH @dir9, @R13+	D	0D	1+2a	----	-	(dir8 × 2) → (R13), R13+2 → (R13)		7.52
DMOVH @R13+, @dir9	D	1D	1+2a	----	-	(R13) → (dir8 × 2), R13+2 → (R13)		7.53
DMOVB @dir8, R13	D	0A	b	----	-	(dir8) → R13	Byte	7.46
DMOVB R13, @dir8	D	1A	a	----	-	R13 → (dir8)		7.47
DMOVB @dir8, @R13+	D	0E	1+2a	----	-	(dir8) → (R13), R13+2 → (R13)		7.48
DMOVB @R13+, @dir8	D	1E	1+2a	----	-	(R13) → (dir8), R13+2 → (R13)		7.49

- The field dir8 in FORMAT_D Instruction format has the following relation to the values of dir8, dir9, dir10 in assembly notation.

  dir8 = dir8

  dir8 = dir9 >> 1

  dir8 = dir10 >> 2

Table A.2-14 Bit Search Instructions (3 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
SRCH0 Ri	E	97-C	1	----	-	search_zero(Ri) → Ri	Searches first 0 Bit	7.110
SRCH1 Ri	E	97-D	1	----	-	search_one(Ri) → Ri	Searches first 1 Bit	7.111
SRCHC Ri	E	97-E	1	----	-	search_change(Ri) → Ri	Searches first change	7.112

Table A.2-15 Other Instructions (16 Instructions)

Mnemonic	Format	OP	CYC	FLAG NZVC	RMW	Operation	Remarks	Reference
NOP	E’	9F-A	1	----	-	No change		7.137
ANDCCR #u8	D	83	1	CCCC	-	CCR & u8 → CCR		7.12
ORCCR #u8	D	93	1	CCCC	-	CCR | u8 → CCR		7.141
STILM #u8	D	87	1	----	-	u8 → ILM	Sets ILM immediate value	7.126
ADDSP #s10	D	A3	1	----	-	R15+s8 × 4 → R15		7.8
EXTSB Ri	E	97-8	1	----	-	exts(Ri[7:0]) → Ri	Sign extension 8 → 32	7.59
EXTUB Ri	E	97-9	1	----	-	extu(Ri[7:0]) → Ri	Zero extension8 → 32	7.61
EXTSH Ri	E	97-A	1	----	-	exts(Ri[15:0]) → Ri	Sign extension 16 → 32	7.60
EXTUH Ri	E	97-B	1	----	-	extu(Ri[15:0]) → Ri	Zero extension16 → 32	7.62
LDM0 (reglist)	D	8C	*1	----	-	for Ri of reglist (R15) → Ri, R15+4 → R15	Load Multiple R0 to R7	7.109
LDM1 (reglist)	D	8D	*1	----	-	for Ri of reglist (R15) → Ri, R15+4 → R15	Load Multiple R8 to R15	7.110
STM0 (reglist)	D	8E	*2	----	-	for Ri of reglist R15-4 → R15, Ri → (R15)	Store multiple R0 to R7	7.127
STM1 (reglist)	D	8F	*2	----	-	for Ri of reglist R15-4 → R15, Ri → (R15)	Store multiple R8 to R15	7.128
ENTER #u10	D	0F	1+a	----	-	R14 → (R15-4) , R15-4 → R14, R15-extu(u8 × 4) → R15	Function entry processing	7.54
LEAVE	E’	9F-9	b	----	-	R14+4 → R15, (R15-4) → R14	Function exit processing	7.85
XCHB @Rj, Ri	A	8A	2a	----	❍	Ri → TEMP, extu((Rj)) → Ri, TEMP → (Rj)	Byte data for semaphore processing	7.132

*1: The number of execution cycles for LDM0(reglist) and LDM1(reglist) is b × n cycles when "n" is the number of registers designated.
*2: The number of execution cycles for STM0(reglist)and STM1(reglist) is a × n when "n" is the number of registers designated.

- In the ADD SP Instruction, the field s8 in TYPR-D Instruction Format has the following relation to the value of s10 in assembly notation.

  s8 = s10 >> 2

- In the ENTER Instruction, the field u8 in TYPR-D Instruction Format has the following relation to the value of u10 in assembly notation.

  u8 = u10 >> 2

Table A.2-16 FPU Memory Load Instructions (7 Instructions)

Mnemonic	Format	OP	CYC	FCC ELGU	RMW	Operation	Remarks	Reference
FLD @Rj, FRi	K	07-C	a	----	-	(Rj) → FRi	Word	7.70
FLD @(R13, Rj), FRi	K	07-E	a	----	-	(R13+Rj) → FRi	Word	7.71
FLD @(R14, disp16), FRi	L	07-D0	a	----	-	(R14+o14 × 4) → FRi	Word	7.72
FLD @(R15, udisp16), FRi	L	07-D4	a	----	-	(R15+u14 × 4) → FRi	Word	7.73
FLD @R15+, FRi	L	07-D8	a	----	-	(R15) → FRi R15 + 4 → R15	Word	7.74
FLD @(BP, udisp18), FRi	J	07-7	a	----	-	(BP+u16 × 4) → FRi	Word	7.75
FLDM (frlist)	N	07-DC	*1	----	-	for FRi of frlist (R15) → FRi R15 + 4 → R15	Load Multiple FR0 to FR15	7.76

*1: The number of execution cycles for FLDM instruction is a × n when "n" is the number of registers designated.

- The field o14 and u14 in TYPE-L instruction format have the following relation to the values disp16, udisp16 in assembly notation.

  o14 = disp16 >> 2

  u14 = udisp16 >> 2

- The field u16 in TYPE-J instruction format has the following relation to the value udisp18 in assembly notation.

  u16 = udisp18 >> 2

Table A.2-17 FPU Memory Store Instructions (7 Instructions)

Mnemonic	Format	OP	CYC	FCC ELGU	RMW	Operation	Remarks	Reference
FST FRi, @Rj	K	17-C	a	----	-	FRi → (Rj)	Word	7.83
FST FRi, @(R13, Rj)	K	17-E	a	----	-	FRi → (R13+Rj)	Word	7.84
FST FRi, @(R14, disp16)	L	17-D0	a	----	-	FRi → (R14+o14 × 4)	Word	7.85
FST FRi, @(R15, udisp16)	L	17-D4	a	----	-	FRi → (R15+u14 × 4)	Word	7.86
FST FRi, @-R15	L	17-D8	a	----	-	R15 - 4 → R15 FRi → (R15)	Word	7.87
FST FRi, @(BP, udisp18)	J	17-7	a	----	-	FRi → (BP+u16 × 4)	Word	7.88
FSTM (frlist)	N	17-DC	*1	----	-	for FRi of frlist R15 - 4 → R15 FRi → (R15)	Store Multiple FR0 to FR15	7.89

*1: The number of execution cycles for FSTM instruction is a × n when "n" is the number of registers designated.

- The field o14 and u14 in TYPE-L instruction format have the following relation to the values disp16 and udisp16 in assembly notation.

  o14 = disp16 >> 2

  u14 = udisp16 >> 2

- The field u16 in TYPE-J instruction format has the following relation to the value udisp18 in assembly notation.

  u16 = udisp18 >> 2

Table A.2-18 FPU Single-Precision Floating Point Calculation Instruction (12 Instructions)

Mnemonic	Format	OP	CYC	FCC ELGU	RMW	Operation	Remarks	Reference
FADDs FRk, FRj, FRi	M	07-A0	1	----	-	FRk + FRj → FRi		7.64
FSUBs FRk, FRj, FRi	M	07-A2	1	----	-	FRk - FRj → FRi		7.91
FCMPs FRk, FRj	M	07-A4	1	CCCC	-	FRk - FRj		7.67
FMULs FRk, FRj, FRi	M	07-A7	1	----	-	FRk × FRj → FRi		7.80
FDIVs FRk, FRj, FRi	M	07-AA	9	----	-	FRk / FRj → FRi		7.68
FNEGs FRj, FRi	M	07-AF	1	----	-	FRj × -1 → FRi		7.81
FABSs FRj, FRi	M	07-AC	1	----	-	|FRj| → FRi		7.63
FMADDs FRk, FRj, FRi	M	07-A5	4	----	-	FRk × FRj + FRi → FRi		7.77
FMSUBs FRk, FRj, FRi	M	07-A6	4	----	-	FRk × FRj - FRi → FRi		7.79
FSQRTs FRj, FRi	M	07-AB	14	----	-	$\sqrt{FRj}$ → FRi		7.82
FiTOs FRj, FRi	M	07-A8	1	----	-	(float)FRj → FRi		7.69
FsTOi FRj, FRi	M	07-A9	1	----	-	(int)FRj → FRi		7.90

Table A.2-19 FPU Inter-Register Transfer Instruction (3 Instructions)

Mnemonic	Format	OP	CYC	FCC ELGU	RMW	Operation	Remarks	Reference
FMOVs FRj, FRi	M	07-AE	1	----	-	FRj → FRi		7.78
MOV Rj, FRi	K	07-3	1	----	-	Rj → FRi		7.131
MOV FRj, Ri	K	17-3	1	----	-	FRj → Ri		7.132

Table A.2-20 FPU Branching Instruction without Delay (16 Instructions)

Mnemonic	Format	OP	CYC	FCC ELGU	RMW	Operation	Remarks	Reference
FBN	N	07-F0	2	----	-	No branch		7.65
FBA label17	N	07-FF	2	----	-	PC+4+exts(rel16 × 2) → PC		7.65
FBNE label17	N	07-F7	2	----	-	if (L || G || U) then PC+4+exts(rel16 × 2) → PC		7.65
FBE label17	N	07-F8	2	----	-	if (E) then PC+4+exts(rel16 × 2) → PC		7.65
FBLG label17	N	07-F6	2	----	-	if (L || G) then PC+4+exts(rel16 × 2) → PC		7.65
FBUE label17	N	07-F9	2	----	-	if (E || U) then PC+4+exts(rel16 × 2) → PC		7.65
FBUL label17	N	07-F5	2	----	-	if (L || U) then PC+4+exts(rel16 × 2) → PC		7.65
FBGE label17	N	07-FA	2	----	-	if (G || E) then PC+4+exts(rel16 × 2) → PC		7.65
FBL label17	N	07-F4	2	----	-	if (L) then PC+4+exts(rel16 × 2) → PC		7.65
FBUGE label17	N	07-FB	2	----	-	if (U || G || E) then PC+4+exts(rel16 × 2) → PC		7.65
FBUG label17	N	07-F3	2	----	-	if (U || G) then PC+4+exts(rel16 × 2) → PC		7.65
FBLE label17	N	07-FC	2	----	-	if (L || E) then PC+4+exts(rel16 × 2) → PC		7.65
FBG label17	N	07-F2	2	----	-	if (G) then PC+4+exts(rel16 × 2) → PC		7.65
FBULE label17	N	07-FD	2	----	-	if (E || L || U) then PC+4+exts(rel16 × 2) → PC		7.65
FBU label17	N	07-F1	2	----	-	if (U) then PC+4+exts(rel16 × 2) → PC		7.65
FBO label17	N	07-FE	2	----	-	if (E || L || G) then PC+4+exts(rel16 × 2) → PC		7.65

- The field rel16 in TYPE-N instruction format has the following relation to the value label17 in assembly notation.

  rel16 = (label17-PC-4)/2

FR81 Family

Table A.2-21 FPU Branching Instruction with Delay (16 Instructions)

Mnemonic	Format	OP	CYC	FCC ELGU	RMW	Operation	Remarks	Reference
FBN:D	N	17-F0	2	----	-	No branch		7.66
FBA:D label17	N	17-FF	2	----	-	PC+4+exts(rel16 × 2) → PC		7.66
FBNE:D label17	N	17-F7	2	----	-	if (L || G || U) then PC+4+exts(rel16 × 2) → PC		7.66
FBE:D label17	N	17-F8	2	----	-	if (E) then PC+4+exts(rel16 × 2) → PC		7.66
FBLG:D label17	N	17-F6	2	----	-	if (L || G) then PC+4+exts(rel16 × 2) → PC		7.66
FBUE:D label17	N	17-F9	2	----	-	if (E || U) then PC+4+exts(rel16 × 2) → PC		7.66
FBUL:D label17	N	17-F5	2	----	-	if (L || U) then PC+4+exts(rel16 × 2) → PC		7.66
FBGE:D label17	N	17-FA	2	----	-	if (G || E) then PC+4+exts(rel16 × 2) → PC		7.66
FBL:D label17	N	17-F4	2	----	-	if (L) then PC+4+exts(rel16 × 2) → PC		7.66
FBUGE:D label17	N	17-FB	2	----	-	if (U || G || E) then PC+4+exts(rel16 × 2) → PC		7.66
FBUG:D label17	N	17-F3	2	----	-	if (U || G) then PC+4+exts(rel16 × 2) → PC		7.66
FBLE:D label17	N	17-FC	2	----	-	if (L || E) then PC+4+exts(rel16 × 2) → PC		7.66
FBG:D label17	N	17-F2	2	----	-	if (G) then PC+4+exts(rel16 × 2) → PC		7.66
FBULE:D label17	N	17-FD	2	----	-	if (E || L || U) then PC+4+exts(rel16 × 2) → PC		7.66
FBU:D label17	N	17-F1	2	----	-	if (U) then PC+4+exts(rel16 × 2) → PC		7.66
FBO:D label17	N	17-FE	2	----	-	if (E || L || G) then PC+4+exts(rel16 × 2) → PC		7.66

- Delayed Branching Instructions are branched after always executing the following Instruction (the Delay Slot).
- The field rel16 in TYPE-N instruction format has the following relation to the value label17 in assembly notation.

  rel16 = (label17-PC-4)/2

A.3 List of Instructions that can be positioned in the Delay Slot

This section shows the Instructions List that can be positioned in the delay slot of Delay Branching Instruction.

- Add/Subtract Instructions

ADD Rj, Ri
ADD #14, Ri
ADD2 #i4, Ri
ADDC Rj, Ri
ADDN Rj, Ri
ADDN #i4, Ri
ADDN2 #i4, Ri
SUB Rj, Ri
SUBC Rj, Ri
SUBN Rj, Ri

- Compare Calculation Instructions

CMP Rj, Ri
CMP #i4, Ri
CMP2 #i4, Ri

- Logical Calculation Instructions

AND Rj, Ri
OR Rj, Ri
EOR Rj, Ri

- Multiply/ Divide Instructions

DIV0S Ri
DIV0U Ri
DIV1 Ri
DIV2 Ri
DIV3
DIV4S

- Shift Instructions

LSL Rj, Ri
LSL #u4, Ri
LSL2 #u4, Ri
LSR Rj, Ri
LSR #u4, Ri
LSR2 #u4, Ri
ASR Rj, Ri
ASR #u4, Ri
ASR2 #u4, Ri

- Immediate Data Transfer Instructions

LDI:8 #i8, Ri

- Memory Load Instructions

LD @Rj, Ri
LD @(R13, Rj), Ri
LD @(R14, disp10), Ri
LD @(R15, udisp6), Ri
LD @R15+, Ri
LD @R15+, Rs

LDUH @Rj, Ri	LDUH @(R13, Rj), Ri	LDUH @(R14, disp9), Ri
LDUB @Rj, Ri	LDUB @(R13, Rj), Ri	LDUB @(R14, disp8), Ri

- Memory Store Instructions

ST Ri, @Rj	ST Ri, @(R13, Rj)	ST Ri, @(R14, disp10)
ST Ri, @(R15, udisp6)	ST Ri, @-R15	ST Rs, @-R15
ST PS, @-R15		
STH Ri, @Rj	STH Ri, @(R13, Rj)	STH Ri, @(R14, disp9)
STB Ri, @Rj	STB Ri, @(R13, Rj)	STB Ri, @(R14, disp8)

- Inter-Register Transfer Instructions

MOV Rj, Ri	MOV Rs, Ri	MOV Ri, Rs
MOV PS, Ri	MOV Ri, PS	

- Direct Addressing Instructions

DMOV @dir10, R13	DMOV R13, @dir10	DMOVH @dir9, R13
DMOVH R13, @dir9	DMOVB @dir8, R13	DMOVB R13, @dir8

- Bit Search Instructions

SRCH0 Ri	SRCH1 Ri	SRCHC Ri

- Other Instructions

NOP	ANDCCR #u8	ORCCR #u8
STILM #u8	ADDSP #s10	EXTSB Ri
EXTUB Ri	EXTSH Ri	EXTUH Ri
LEAVE		

APPENDIX B Instruction Maps

It includes instruction maps of FR81 Family CPU.

B.1 Instruction Maps
B.2 Extension Instruction Maps

FR81 Family

B.1 Instruction Maps

The following shows an instruction map when the operation code consists of 8 or less bits.

Figure B.1-1 illustrates in tabular form 8 bit operation codes (OP) for each instruction. Instructions where operation code (OP) is less than 8 bit, they have been converted into 8 bit by packing them on MSB side.

Figure B.1-1 Instruction Map

Higher 4 bits (columns) / Lower 4 bits (rows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	LD @(R13,Rj), Ri	ST Ri, @(R13,Rj)	LD @(R14, disp10),Ri	ST Ri,@ (R14, disp10)	LDUH @(R14, disp9),Ri	STH Ri,@(R14, disp9)	LDUB @(R14, disp8),Ri	STB Ri,@(R14, disp8)	BANDL #u4,@Ri	BORL #u4,@Ri	ADDN# i4,Ri	LSR #u4,Ri	LDI:8 #i8,Ri	CALL label12	BRA label9	BRA:D label9
1	LDUH @(R13,Rj), Ri	STH Ri, @(R13,Rj)							BANDH #u4,@Ri	BORH #u4,@Ri	ADDN2 #i4,Ri	LSR2 #u4,Ri			BNO label9	BNO:D label9
2	LDUB @(R13,Rj), Ri	STB Ri, @(R13,Rj)							AND Rj, Ri	OR Rj, Ri	ADDN Rj,Ri	LSR Rj, Ri			BEQ label9	BEQ:D label9
3	LD @(R15, udisp6),Ri	ST Ri, @(R15,ud6)							ANDCCR #u8	ORCCR #u8	ADDSP #s10	MOV Ri, Rs			BNE label9	BNE:D label9
4	LD @Rj,Ri	ST Ri,@ Rj							AND Rj,@Ri	OR Rj,@Ri	ADD #i4,Ri	LSL #u4,Ri			BC label9	BC:D label9
5	LDUH @Rj,Ri	STH Ri,@Rj							ANDH Rj,@Ri	ORH Rj,@Ri	ADD2 #i4,Ri	LSL2 #u4,Ri			BNC label9	BNC:D label9
6	LDUB @Rj,Ri	STB Ri,@Rj							ANDB Rj,@Ri	ORB Rj,@Ri	ADD Rj,Ri	LSL Rj,Ri			BN label9	BN:D label9
7	Refer to APPENDIX B.2	Refer to APPENDIX B.2							STILM #u8	Refer to APPENDIX B.2	ADDC Rj, Ri	MOV Rs, Ri			BP label9	BP:D label9
8	DMOV @d10,R13	DMOV R13,@d10							BTSTL #u4,@Ri	BEORL #u4,@Ri	CMP #i4,Ri	ASR #u4,Ri		CALL:D label12	BV label9	BV:D label9
9	DMOVH @d9, R13	DMOVH R13,@d 9							BTSTH #u4,@Ri	BEORH #u4,@Ri	CMP2 #i4,Ri	ASR2 #u4,Ri			BNV label9	BNV:D label9
A	DMOVB@ d8, R13	DMOVB R13,@d 8							XCHB @Rj,Ri	EOR Rj,Ri	CMP Rj,Ri	ASR Rj,Ri			BLT label9	BLT:D label9
B	DMOV @d10,@–R15	DMOV @R15+,@d10							MOV Rj, Ri	LD:20 #i20,Ri	MULU Rj, Ri	MULUH Rj, Ri			BGE label9	BGE:D label9
C	DMOV @d10,@R13+	DMOV @R13+,@d10							LDM0 (reglist)	EOR Rj, @ Ri	SUB Rj, Ri	LDRES @Ri+,#u4			BLE label9	BLE:D label9
D	DMOVH @d9, @R13+	DMOVH @R13+, @ d9							LDM1 (reglist)	EORH Rj,@Ri	SUBC Rj, Ri	STRES #u4,@Ri+			BGT label9	BGT:D label9
E	DMOVB @d8, @R13+	DMOVB @R13+, @ d8							STM0 (reglist)	EORB Rj,@Ri	SUBN Rj,Ri				BLS label9	BLS:D label9
F	ENTER #u10	INT #u8							STM1 (reglist)	Refer to APPENDIX B.2	MUL Rj,Ri	MULH Rj,Ri			BHI label9	BHI:D label9

B.2 Extension Instruction Maps

The following shows an instruction map when the operation code consists of 12 or more bits.

Instructions with a 12-bit operation code (OP), which is divided into 8 higher bits and 4 lower bits are shown in Table B.2-1.

Table B.2-1 Instruction Map with 12-Bit Operation Code

Higher 8 bits (columns) / Lower 4 bits (rows)

	07	17	97	9F
0	LD @R15+,Ri	ST Ri,@-R15	JMP @Ri	JMP:D @Ri
1	MOV Ri,PS	MOV PS,Ri	CALL @Ri	CALL:D @Ri
2	LCALL label21	LCALL:D label21	RET	RET:D
3	MOV Rj, FRi	MOV FRi, Rj	RETI	INTE
4	LD @(BP, udisp18), Ri	ST Ri, @(BP, udisp18)	DIV0S Ri	-
5	LDUH @(BP, udisp17), Ri	STH Ri, @(BP, udisp17)	DIV0U Ri	-
6	LDUB @(BP, udisp16), Ri	STB Ri, @(BP, udisp16)	DIV1 Ri	DIV3
7	FLD @(BP, udisp18), FRi	FST FRi, @(BP, udisp18)	DIV2 Ri	DIV4S
8	LD @R15+,Rs	ST Rs,@-R15	EXTSB Ri	LDI:32 #i32,Ri
9	LD @R15+,PS	ST PS,@-R15	EXTUB Ri	LEAVE
A	Refer to Table B.2-2	Refer to Table B.2-2	EXTSH Ri	NOP
B	-	-	EXTUH Ri	-
C	FLD @Rj, FRi	FST FRi, @Rj	-	SRCH0
D	Refer to Table B.2-2	Refer to Table B.2-2	-	SRCH1
E	FLD @(R13, Rj), FRi	FST FRi, @(R13, Rj)	-	SRCHC
F	Refer to Table B.2-3	Refer to Table B.2-3	-	-

-: Undefined

FR81 Family

Instructions with 16-bit and 14-bit operation codes (OP), each of which is divided into 8 higher bits and 8 lower bits are shown in Tables Table B.2-2 and Table B.2-3. An instruction with a 14-bit operation code is padded to the MSB side to convert the 14-bit operation code to a 16-bit operation code.

Table B.2-2 Instruction Map with 16-Bit/14-Bit Operation Code

Higher 8 bits (columns); Lower 8 bits (rows)

	07	17
A0	FADDs FRK, FRj, FRi	-
A1	-	-
A2	FSUBs FRK, FRj, FRi	-
A3	-	-
A4	FCMPs FRk, FRj	-
A5	FMADDs FRk, FRj, FRi	-
A6	FMSUBs FRk, FRj, FRi	-
A7	FMULs FRK, FRj, FRi	-
A8	FiTOs FRj, FRi	-
A9	FsTOi FRj, FRi	-
AA	FDIVs FRK, FRj, FRi	-
AB	FSQRTs FRk, FRj	-
AC	FABSs FRj, FRi	-
AD	-	-
AE	FMOVs FRj, FRi	-
AF	FNEGs FRj, FRi	-
D0*	FLD @(R14, disp16), FRi	FST FRi, @(R14, disp16)
D4*	FLD @(R15, udisp16), FRi	FST FRi, @(R15, udisp16)
D8*	FLD @R15+, FRi	FST FRi, @-R15
DC*	FLDM (frlist)	FSTM (frlist)

*: Instruction of 14-Bit Operation Code
-: Undefined

Table B.2-3 Instruction Map with 16-Bit Operation Code

Higher 8 bits (columns) / Lower 8 bits (rows)

	07	17
F0	FBN	FBN:D
F1	FBU label17	FBU:D label17
F2	FBG label17	FBG:D label17
F3	FBUG label17	FBUG:D label17
F4	FBL label17	FBL:D label17
F5	FBUL label17	FBUL:D label17
F6	FBLG label17	FBLG:D label17
F7	FBNE label17	FBNE:D label17
F8	FBE label17	FBE:D label17
F9	FBUE label17	FBUE:D label17
FA	FBGE label17	FBGE:D label17
FB	FBUGE label17	FBUGE:D label17
FC	FBLE label17	FBLE:D label17
FD	FBULE label17	FBULE:D label17
FE	FBO label17	FBO:D label17
FF	FBA label17	FBA:D label17

APPENDIX C Supplemental Explanation about FPU Exception Processing

C.1 Conformity with IEEE754-1985 Standard

The FR81 Family CPU conforms to the IEEE754-1985 standard (hereinafter referred to as "IEEE754"), excluding the following.

1. Overflow

   IEEE754 defines that the biased (+192 for single-precision) exponent part is used as the result if overflow occurs; however, this architecture does not provide for certain cases such as the addition of the biased value.

2. Underflow

   IEEE754 defines that the biased (-192 for single-precision) exponent part is used as the result if underflow occurs; however, this architecture does not provide for certain cases such as the reduction of the biased value.

   When the result is an unnormalized number, the result is flushed to zero.

3. Unnormalized number

   IEEE754 defines an unnormalized number to prevent the result from being flushed rapidly to zero; however, the unnormalized number is not supported in this architecture. If an unnormalized number is input when the unnormalized number input exception is prohibited, its numeric value is assumed to be zero for calculation purposes. This is also applied when the calculation result is an unnormalized number; therefore, the result is set to zero.

4. QNaN

   The QNaN output pattern is fixed to $7FFFFFFF_H$, excluding the move, sign inversion, and absolute value instructions.

5. Unsupported instructions

   This architecture does not support the following instructions.

   - Remainder
   - Integer rounding (Round Floating-Point Number to Integer Value)
   - Conversion between binary and decimal numbers

6. Floating point calculation exception

   This exception occurs when the sufficient calculation result is not obtained for IEEE754. This architecture provides six exceptions in total: five exceptions (Inexact, Underflow, Overflow, Division by Zero, and Invalid Calculation), which are defined in IEEE754, as well as one exception (unnormalized number input).

   In IEEE754, if a floating point calculation exception occurs, the defined operation is carried out to generate a trap. In this architecture, it is provided as an exception; therefore, no data is written to the destination when a floating point calculation exception occurs.

C.2 FPU Exceptions

FPU exceptions are classified into six types: five exceptions (Inexact, Underflow, Overflow, Division by Zero, and Invalid Calculation), which are defined in IEEE754, and an exception that is generated when an unnormalized number is input. Whether or not to generate those calculation exceptions can be specified using the FPU control register (FCR.EEF). The result output upon the exception conditions being satisfied varies depending on the FCR.EEF setting.

If an exception is not permitted, the result of each calculation is output so that the requested result is obtained when calculation runs on even if an exception occurs, or so that it can be recognized from the result that the calculation is invalid. When an exception is permitted, if the significant calculation result cannot be obtained due to an exception factor, the calculation result is not written; otherwise, it is written.

1. Invalid calculation exception (Invalid Operation)

   This exception occurs when calculation cannot be carried out properly because the specified operand is invalid for the calculation. In concrete terms, this exception occurs when:

   - SNaN has been input;
   - the result is in infinite format ($\infty - \infty$, $0 \times \infty$, 0/0, ∞/∞, $\sqrt{-1}$ etc.)
   - the conversion source value is not indicated in the conversion destination format using a conversion instruction.

   If this exception occurs, the following operations are carried out.

   [FCR:EEF:V=1]

   Writing to the floating point register or FCR:FCC is prohibited.

   The FCR.CEF.V flag is set to generate an FPU exception.

   [FCR:EEF:V=0]

   QNaN ($7FFFFFFF_H$) is stored in the floating point register for instructions other than conversion or comparison instructions.

   For the conversion instruction, ± MAX is stored in the floating point register.

   For the comparison instruction, the FCR:FCC:U flag is set.

   The FCR:ECF:V flag is set.

2. Division-by-Zero exception (Division by Zero)

   This exception occurs when performing division by zero. If this exception occurs, the following operations are carried out.

   [FCR:EEF:Z=1]

   Writing to the floating point register is prohibited.

   The FCR:CEF.Z flag is set to generate this exception.

   [FCR:EEF:Z=0]

   The infinity is stored in the floating point register. If the sign is the same, it is set to a positive sign. If the sign is different, it is set to a negative sign.

   The FCR:ECF:Z flag is set.

3. Overflow exception (Overflow)

This exception occurs when, during calculation, the exponent exceeds the maximum that can be expressed in the specified format. If this exception occurs, the following operations are carried out.

[FCR:EEF:O=1]

Writing to the floating point register is prohibited.

The FCR:CEF:O flag is set to generate this exception.

[FCR:EEF:O=0]

As shown in the following Table C.2-1, the value is written to the floating point register based on the rounding mode.

The FCR:ECF:O flag is set.

Table C.2-1 Output Result in Rounding Mode and at Overflow

Rounding mode (FCR:RM)	Output result	
	Positive overflow	Negative overflow
00_B (Latest value)	+∞	-∞
01_B (Zero)	+MAX	-MAX
10_B (+∞)	+∞	-MAX
11_B (-∞)	+MAX	-∞

4. Underflow exception (Underflow)

This exception occurs when the rounding result is incorrect and the absolute value is less than the minimum normalized number in the specified format. In concrete terms, this exception occurs when the exponent is set to 0 while normalizing a significand or when the exponent is set to a negative value during multiplication or division. This exception also occurs when a specific value is set to the minimum normalized number after round processing while it is an unnormalized number before round processing (pre-rounding rule). This exception does not occur when the result of non-round processing is correct and set to zero. If this exception occurs, the following operations are carried out.

[FCR:EEF:U=1]

Writing to the floating point register is prohibited.

The FCR:CEF:U flag is set to generate this exception.

[FCR:EEF:U=0]

Zero is always stored in the floating point register. (Zero flush)

The FCR:ECF:U flag is set.

5. Inexact exception (Inexact)

This exception occurs when the rounding result is incorrect (including a case where an Underflow exception is detected at FCR:EEF:U = 0 when a unnormalized number is flushed to zero) or when overflow has occurred because an Overflow exception is invalid (including a case where overflow has occurred as a result of round processing). If this exception occurs, the following operations are carried out.

[FCR:EEF:X=1]

Writing to the floating point register is prohibited.

The FCR:CEF:X flag is set to generate this exception.

[FCR:EEF:X=0]

The calculation result is stored in the floating point register.

The FCR:ECF:X flag is set.

6. Unnormalized Number Input exception (Denormalized Number Input)

This exception occurs when an unnormalized number is specified in the input operand. If this exception occurs, the following operations are carried out.

[FCR:EEF:D=1]

Writing to the floating point register is prohibited.

The FCR:CEF:D flag is set to generate this exception.

[FCR.EEF.D=0]

The input operand that is set to an unnormalized number is flushed to zero before calculation.

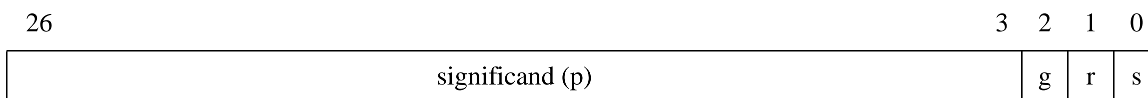
The FCR:ECF:D flag is set.

C.3 Round Processing

Rounding processing conforms to IEEE754. A significand is expressed by 24 bits (single-precision). If a significand of calculation result is expressed by the number of bits greater than 24 bits (including unnormalized numbers), round processing is performed to obtain the approximate value of 24 bits (single-precision). The following explains round processing.

The calculation result (S) of the significand is defined as follows (see next if the guard bit is omitted (already processed)).

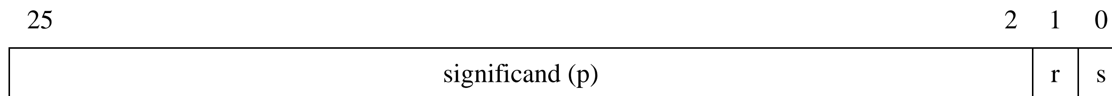
Figure C.3-1 Calculation Result of Significand including Guard Bit



Here, "g" indicates a guard bit, "r" indicates a round bit, "s" indicates a sticky bit, and "p" indicates a significand.

Next, obtain the OR value (r∪s) between "r" and "s". Then set the result as "s" and set "g" as "r" again.

Figure C.3-2 Calculation Result of Significand omitting Guard Bit



Perform round processing based on the following Table C.3-1. Here, "S" indicates the calculation result of the significand, "r" indicates a round bit, "s" indicates a sticky bit, and "LSB" indicates the LSB of "p". ("!s" indicates the reversal value of "s".)

Table C.3-1 Rounding Mode and Rounding Processing of Significand

Rounding mode (FCR.RM)	S ≥ 0	S < 0
00_B (Latest value)	"p+1" if "r^!s^LSB" or "r^s" is true	"p+1" if "r^!s^LSB" or "r^s" is true
01_B (Zero)		
10_B (+∞)	"p+1" if "r∪s" is true	
11_B (-∞)		"p+1" if "r∪s" is true

The blank column means that the "p" value is used as the result.

INDEX

The index follows on the next page.
This is listed in alphabetic order.

Index

Numerics

20-bit Addressing
- 20-bit Addressing Area & 32-bit Addressing Area 11

32-bit Addressing
- 20-bit Addressing Area & 32-bit Addressing Area 11

A

Access
- Data Access 14
- Program Access 14

ADD
- ADD (Add 4bit Immediate Data to Destination Register)....105
- ADD (Add Word Data of Source Register to Destination Register) 107
- ADD2 (Add 4bit Immediate Data to Destination Register)....109

ADDC
- ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) 111

ADDN
- ADDN (Add Immediate Data to Destination Register) 113
- ADDN (Add Word Data of Source Register to Destination Register) 115
- ADDN2 (Add Immediate Data to Destination Register)....117

Address Space
- Address Space....8

Addressing
- 20-bit Addressing Area & 32-bit Addressing Area 11

Addressing Formats
- Addressing Formats....86

ADDSP
- ADDSP (Add Stack Pointer and Immediate Data) 119

Alignment
- Word Alignment 14

AND
- AND (And Word Data of Source Register to Data in Memory)....121
- AND (And Word Data of Source Register to Destination Register) 123

ANDB
- ANDB (And Byte Data of Source Register to Data in Memory)....125

ANDCCR
- ANDCCR (And Condition Code Register and Immediate Data)....127

ANDH
- ANDH (And Halfword Data of Source Register to Data in Memory)....129

Arithmetic shift
ASR (Arithmetic shift to the Right Direction) 131, 133
ASR2 (Arithmetic shift to the Right Direction) 135

ASR
ASR (Arithmetic shift to the Right Direction) 131, 133
ASR2 (Arithmetic shift to the Right Direction) 135

B

BANDH
BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory)....137

BANDL
BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory)....139

Bcc
Bcc (Branch relative if Condition satisfied)....141

Bcc:D
Bcc:D (Branch relative if Condition satisfied) 143

BEORH
BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory)....145

BEORL
BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory)....147

BORH
BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory) 149

BORL
BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory) 151

Branch
Bcc (Branch relative if Condition satisfied)....141
Bcc:D (Branch relative if Condition satisfied) 143

Branching
Delayed Branching Instructions 97
Delayed branching processing....78
Example of branching with non-delayed branching instructions 78
Example of processing of delayed branching instruction....79
Non-Delayed Branching Instructions....99
Specific example of Delayed Branching Instructions 98

Branching Instructions
Branching Instructions and Delay Slot....97

BTSTH
BTSTH (Test Higher 4bit of Byte Data in Memory) 153

BTSTL
BTSTL (Test Lower 4bit of Byte Data in Memory) 155

Bypassing
Register Bypassing....74

Byte Data
ANDB (And Byte Data of Source Register to Data in Memory)....125
BTSTH (Test Higher 4bit of Byte Data in Memory) 153
BTSTL (Test Lower 4bit of Byte Data in Memory) 155
Byte Data 12
DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) 199
DMOVB (Move Byte Data from Direct Address to Register)....195
DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) 201
DMOVB (Move Byte Data from Register to Direct Address)....197
EORB (Exclusive Or Byte Data of Source Register to Data in Memory)....217
EXTSB (Sign Extend from Byte Data to Word Data) 221
EXTSH (Sign Extend from Byte Data to Word Data) 223
EXTUB (Unsign Extend from Byte Data to Word Data)....225
EXTUH (Unsign Extend from Byte Data to Word Data)....227
LDUB (Load Byte Data in Memory to Register) 306, 308, 310
ORB (Or Byte Data of Source Register to Data in Memory)....360
STB (Store Byte Data in Register to Memory) 394, 396, 398
XCHB (Exchange Byte Data)....420

Byte Order
Byte Order....13

C

CALL
CALL (Call Subroutine) 157, 159

CALL:D
CALL:D (Call Subroutine)....161, 163

Carry Bit
ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) 111

CCR
Condition Code Register (CCR) 21, 23
CMP
CMP (Compare Immediate Data and Destination Register) 165
CMP (Compare Word Data in Source Register and Destination Register)...........................167
CMP2 (Compare Immediate Data and Destination Register) 169
Condition Code Register
ANDCCR (And Condition Code Register and Immediate Data) 127
Condition Code Register (CCR) 23
ORCCR (Or Condition Code Register and Immediate Data) 362
Correction
DIV2 (Correction When Remain is 0)................177
DIV3 (Correction When Remain is 0)................179
DIV4S (Correction Answer for Signed Division) 181
CPU
Features of FR80 Family CPU 2
FR80 Family CPU Register Configuration 16

D

Data Access
Data Access 14
Data Structure
Data Structure 12
Dedicated Registers
Configuration of Dedicated Registers 19
Dedicated Registers 19
Delay Slot
Branching Instructions and Delay Slot 97
Delayed Branching
Delayed branching processing.............................78
Delayed Branching Instruction
Delayed Branching Instructions 97
Example of processing of delayed branching instruction..79
Specific example of Delayed Branching Instructions 98
Destination Register
ADD (Add 4bit Immediate Data to Destination Register) 105
ADD2 (Add 4bit Immediate Data to Destination Register) 109
ADDN (Add Immediate Data to Destination Register) 113
ADDN2 (Add Immediate Data to Destination Register) 117
CMP (Compare Immediate Data and Destination Register) 165
CMP2 (Compare Immediate Data and Destination Register)...169
LDI:8 (Load Immediate 8bit Data to Destination Register)...300
Direct Address
Direct Address Area 8
DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) 187
DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) 191
DMOV (Move Word Data from Direct Address to Register)...183
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 189
DMOV (Move Word Data from Register to Direct Address) 185
DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) 199
DMOVB (Move Byte Data from Direct Address to Register)...195
DMOVB (Move Byte Data from Register to Direct Address) 197
DMOVH (Move Halfword Data from Direct Address to Register) 203
DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) 207
DIV
DIV0S (Initial Setting Up for Signed Division) 171
DIV0U (Initial Setting Up for Unsigned Division) 173
DIV1 (Main Process of Division).......................175
DIV2 (Correction When Remain is 0) 177
DIV3 (Correction When Remain is 0) 179
DIV4S (Correction Answer for Signed Division) 181
Division
DIV0S (Initial Setting Up for Signed Division) 171
DIV0U (Initial Setting Up for Unsigned Division) 173
DIV1 (Main Process of Division).......................175
DIV4S (Correction Answer for Signed Division) 181
Signed Division..100
Step Division Instructions 100
Unsigned Division...101

DMOV
DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) 187
DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) 191
DMOV (Move Word Data from Direct Address to Register) 183
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 189, 193
DMOV (Move Word Data from Register to Direct Address) 185
DMOVB
DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) 199
DMOVB (Move Byte Data from Direct Address to Register) 195
DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) 201
DMOVB (Move Byte Data from Register to Direct Address) 197
DMOVH
DMOVH (Move Halfword Data from Direct Address to Register) 203
DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) 207
DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) 209
DMOVH (Move Halfword Data from Register to Direct Address) 205

E

EIT
Basic Operations in EIT Processing 43
EIT Processing Sequence 44
Multiple EIT Processing 60
Multiple EIT processing and Priority Levels 60
Priority Levels of EIT Requests 61
Recovery from EIT Processing 45
Types of EIT Processing and Prior Preparation 43
Emulator
INTE (Software Interrupt for Emulator) 273
ENTER
ENTER (Enter Function) 211
EOR
BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory) 145
BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory) 147
EOR (Exclusive Or Word Data of Source Register to Destination Register) 215
EOR (Exclusive Or Word Data of Source Register to Data in Memory) 213
EORB
EORB (Exclusive Or Byte Data of Source Register to Data in Memory) 217
EORH
EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) 219
Exception
Exception Processing 48
Exchange
XCHB (Exchange Byte Data) 420
Exclusive Or
EOR (Exclusive Or Word Data of Source Register to Destination Register) 215
EOR (Exclusive Or Word Data of Source Register to Data in Memory) 213
EORB (Exclusive Or Byte Data of Source Register to Data in Memory) 217
EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) 219
Extend
EXTSB (Sign Extend from Byte Data to Word Data) 221
EXTSH (Sign Extend from Byte Data to Word Data) 223
EXTUB (Unsign Extend from Byte Data to Word Data) 225
EXTSB
EXTSB (Sign Extend from Byte Data to Word Data) 221
EXTSH
EXTSH (Sign Extend from Byte Data to Word Data) 223
EXTUB
EXTUB (Unsign Extend from Byte Data to Word Data) 225
EXTUH
EXTUH (Unsign Extend from Byte Data to Word Data) 227

F

FABSs
FABSs (Single Precision Floating Point Absolute Value) 229
FADDs
FADDs (Single Precision Floating Point Add) 230

FBcc
- FBcc (Floating Point Conditional Branch)...........232
- FBcc:D (Floating Point Conditional Branch with Delay Slot) 234

FCMPs
- FCMPs (Single Precision Floating Point Compare) 236

FDIVs
- FDIVs (Single Precision Floating Point Division) 238

First One bit
- SRCH1 (Search First One bit position distance From MSB) 375

First Zero bit
- SRCH0 (Search First Zero bit position distance From MSB) 373

FiTOs
- FiTOs (Convert from Integer to Single Precision Floating Point)....................................240

flag
- Timing when the interrupt enable flag (I) is requested 63

FLD
- FLD (Load Word Data in Memory to Floating Register) 247
- FLD (Single Precision Floating Point Data Load) 242, 243, 244, 245, 246

FLDM
- FLDM (Single Precision Floating Point Data Load to Multiple Register)................................248

FMADDs
- FMADDs (Single Precision Floating Point Multiply and Add)...250

FMOVs
- FMOVs (Single Precision Floating Point Move) 252

FMSUBs
- FMSUBs (Single Precision Floating Point Multiply and Subtract).....................................253

FMULs
- FMULs (Single Precision Floating Point Multiply) 255

FNEGs
- FNEGs (Single Precision Floating Point sign reverse) 257

Format
- Instruction Formats..87

Formats
- Addressing Formats 86
- Instructions Formats 85
- Instructions Notation Formats 85

FR Family
- Changes from the earlier FR Family 4

FR81 Family
- Features of FR81 Family CPU 2
- FR81 Family CPU Register Configuration............16

FSQRTs
- FSQRTs (Single Precision Floating Point Square Root)...258

FST
- FST (Single Precision Floating Point Data Store) 259, 260, 261, 262, 263
- FST (Store Word Data in Floating Point Register to Memory) 264

FSTM
- FSTM (Single Precision Floating Point Data Store from Multiple Register) 265

FsTOi
- FsTOi (Convert from Single Precision Floating Point to Integer)..267

FSUBs
- FSUBs (Single Precision Floating Point Subtract) 269

G

General Interrupt
- General interrupts...53

General-purpose Registers
- Configuration of General-purpose Registers..........17
- General-purpose Registers..................................17
- Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) 75
- Special Usage of General-purpose Registers 18

H

Half Word Data
- Half Word Data...12

Halfword Data
- ANDH (And Halfword Data of Source Register to Data in Memory)................................129
- DMOVH (Move Halfword Data from Direct Address to Register).......................................203
- DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) 207
- DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) 209
- DMOVH (Move Halfword Data from Register to Direct Address)..................................205
- EORH (Exclusive Or Halfword Data of Source Register to Data in Memory) 219
- LDUH (Load Halfword Data in Memory to Register) 313, 315, 317
- MULH (Multiply Halfword Data)......................348

MULUH (Multiply Unsigned Halfword Data) 352
ORH (Or Halfword Data of Source Register to Data in Memory) 364
STH (Store Halfword Data in Register to Memory) 401, 403, 405

hazard
Occurrence of register hazard....................74
Register hazards 74

I

I
Timing when the interrupt enable flag (I) is requested 63

ILM
Interrupt Level Mask Register (ILM)....................22

Immediate 20bit Data
LDI:20 (Load Immediate 20bit Data to Destination Register) 296

Immediate 32 bit Data
LDI:32 (Load Immediate 32 bit Data to Destination Register) 298

Immediate 8bit Data
LDI:8 (Load Immediate 8bit Data to Destination Register) 300

Immediate Data
ADD (Add 4bit Immediate Data to Destination Register) 105
ADD2 (Add 4bit Immediate Data to Destination Register) 109
ADDN (Add Immediate Data to Destination Register) 113
ADDN2 (Add Immediate Data to Destination Register) 117
ADDSP (Add Stack Pointer and Immediate Data) 119
BANDH (And 4bit Immediate Data to Higher 4bit of Byte Data in Memory)....................137
BANDL (And 4bit Immediate Data to Lower 4bit of Byte Data in Memory)....................139
BEORH (Eor 4bit Immediate Data to Higher 4bit of Byte Data in Memory)....................145
BEORL (Eor 4bit Immediate Data to Lower 4bit of Byte Data in Memory)....................147
BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory) 149
BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory) 151
CMP (Compare Immediate Data and Destination Register) 165
CMP2 (Compare Immediate Data and Destination Register) 169
ORCCR (Or Condition Code Register and Immediate Data) 362
STILM (Set Immediate Data to Interrupt Level Mask Register)....................408

Increment Register
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 189, 193
DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) 201

Indirect Address
DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) 187
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 189

Instruction
“INT”Instructions....................57
Branching Instructions and Delay Slot....................97
Delayed Branching Instructions 97
Example of branching with non-delayed branching instructions 78
Example of processing of delayed branching instruction 79
Instruction execution based on Pipeline 70
INTE Instruction 57
Non-Delayed Branching Instructions....................99
Read-Modify-Write type Instructions 96
Specific example of Delayed Branching Instructions 98
Step Division Instructions 100

Instruction Format
Instruction Formats....................87
Instructions Formats 85

Instruction System
Instruction System....................82

Instructions Notation Formats
Instructions Notation Formats....................85

INT
“INT”Instructions....................57
INT (Software Interrupt) 271

INTE
INTE (Software Interrupt for Emulator) 273
INTE Instruction 57

Interlocking
Interlocking 75
Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) 75

Interrupt
General interrupts....................53
INT (Software Interrupt) 271
INTE (Software Interrupt for Emulator) 273
Interrupts....................53

Mismatch in Acceptance and Cancellation of Interrupt 73
Non-maskable Interrupts (NMI) 55
Pipeline Operation and Interrupt Processing 73
Points of Caution while using User Interrupts 66
Preparation while using user interrupts 65
Processing during an Interrupt Processing Routine 66
RETI (Return from Interrupt) 370
Usage Sequence of User Interrupts 65

interrupt enable flag
Timing when the interrupt enable flag (I) is requested 63

Interrupt Level Mask Register
Interrupt Level Mask Register (ILM) 22
Timing of Reflection of Interrupt Level Mask Register (ILM) 64

Interrupt Processing Routine
Processing during an Interrupt Processing Routine 66

J

JMP
JMP (Jump) 275

JMP:D
JMP:D (Jump) 277

Jump
JMP (Jump) 275
JMP:D (Jump) 277

L

LCALL
LCALL (Long Call Subroutine) 279
LCALL:D (Long Call Subroutine) 280

LD
LD (Load Word Data in Memory to Program Status Register) 294
LD (Load Word Data in Memory to Register) 281, 283, 285, 287, 289, 291, 292

LDI:20
LDI:20 (Load Immediate 20bit Data to Destination Register) 296

LDI:32
LDI:32 (Load Immediate 32 bit Data to Destination Register) 298

LDI:8
LDI:8 (Load Immediate 8bit Data to Destination Register) 300

LDM
LDM0 (Load Multiple Registers) 302
LDM1 (Load Multiple Registers) 304

LDUB
LDUB (Load Byte Data in Memory to Register) 306, 308, 310, 312

LDUH
LDUH (Load Halfword Data in Memory to Register) 313, 315, 317, 319

LEAVE
LEAVE (Leave Function) 320

Load
LD (Load Word Data in Memory to Program Status Register) 294
LD (Load Word Data in Memory to Register) 281, 283, 285, 287, 289, 292
LDI:20 (Load Immediate 20bit Data to Destination Register) 296
LDI:32 (Load Immediate 32 bit Data to Destination Register) 298
LDI:8 (Load Immediate 8bit Data to Destination Register) 300
LDM0 (Load Multiple Registers) 302
LDM1 (Load Multiple Registers) 304
LDUB (Load Byte Data in Memory to Register) 306, 308, 310
LDUH (Load Halfword Data in Memory to Register) 313, 315, 317

Logical Shift
LSL (Logical Shift to the Left Direction) 322, 324
LSL2 (Logical Shift to the Left Direction) 326
LSR (Logical Shift to the Right Direction) 328, 330
LSR2 (Logical Shift to the Right Direction) 332

LSL
LSL (Logical Shift to the Left Direction) 322, 324
LSL2 (Logical Shift to the Left Direction) 326

LSR
LSR (Logical Shift to the Right Direction) 328, 330
LSR2 (Logical Shift to the Right Direction) 332

M

MDH
Multiplication/Division Register (MDH, MDL) 30

MDL
Multiplication/Division Register (MDH, MDL) 30

MOV
MOV (Move Word Data in Floating Point Register to General Purpose Register) 345
MOV (Move Word Data in General Purpose Register to Floating Point Register) 344

MOV (Move Word Data in Program Status Register to Destination Register)............................338
MOV (Move Word Data in Source Register to Destination Register)............334, 336, 340
MOV (Move Word Data in Source Register to Program Status Register) 342

Move
MOV (Move Word Data in Program Status Register to Destination Register)............................338
MOV (Move Word Data in Source Register to Destination Register)............334, 336, 340
MOV (Move Word Data in Source Register to Program Status Register) 342

MSB
SRCH0 (Search First Zero bit position distance From MSB) 373
SRCH1 (Search First One bit position distance From MSB) 375

MUL
MUL (Multiply Word Data) 346

MULH
MULH (Multiply Halfword Data) 348

Multiple
Multiple EIT Processing......................................60
Multiple EIT processing and Priority Levels..........60

Multiple Registers
LDM0 (Load Multiple Registers) 302
LDM1 (Load Multiple Registers) 304
STM0 (Store Multiple Registers) 410
STM1 (Store Multiple Registers) 412

Multiplication/Division Register
Multiplication/Division Register (MDH, MDL) 30

Multiply
MUL (Multiply Word Data) 346
MULH (Multiply Halfword Data) 348
MULU (Multiply Unsigned Word Data) 350
MULUH (Multiply Unsigned Halfword Data) 352

MULU
MULU (Multiply Unsigned Word Data) 350

MULUH
MULUH (Multiply Unsigned Halfword Data) 352

N

NMI
Non-maskable Interrupts (NMI) 55

No Operation
NOP (No Operation)...354

Non-block loading
Non-block loading 77

non-delayed branching
Example of branching with non-delayed branching instructions 78

Non-Delayed Branching Instructions
Non-Delayed Branching Instructions....................99

Non-maskable Interrupts
Non-maskable Interrupts (NMI)...........................55

NOP
NOP (No Operation)...354

O

OR
BORH (Or 4bit Immediate Data to Higher 4bit of Byte Data in Memory)................................149
BORL (Or 4bit Immediate Data to Lower 4bit of Byte Data in Memory)................................151
OR (Or Word Data of Source Register to Data in Memory)...356
OR (Or Word Data of Source Register to Destination Register)..358

ORB
ORB (Or Byte Data of Source Register to Data in Memory)...360

ORCCR
ORCCR (Or Condition Code Register and Immediate Data)...362

ORH
ORH (Or Halfword Data of Source Register to Data in Memory).......................................364

P

PC
Program Counter (PC) 20

Pipeline
How to prevent mismatched pipeline conditions? 73
Instruction execution based on Pipeline 70
Pipeline Operation and Interrupt Processing..........73

Pointer
ADDSP (Add Stack Pointer and Immediate Data) 119

Post
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 193
DMOVB (Move Byte Data from Direct Address to Post Increment Register Indirect Address) 199
DMOVB (Move Byte Data from Post Increment Register Indirect Address to Direct Address) 201

DMOVH (Move Halfword Data from Direct Address to Post Increment Register Indirect Address) 207
DMOVH (Move Halfword Data from Post Increment Register Indirect Address to Direct Address) 209

Prior Preparation
Types of EIT Processing and Prior Preparation 43

Priority Levels
Multiple EIT processing and Priority Levels 60
Priority Levels of EIT Requests 61

Processing
Multiple EIT Processing....60

processing
Multiple EIT processing and Priority Levels 60

Program Access
Program Access....14

Program Counter
Program Counter (PC)....20

Program Status
LD (Load Word Data in Memory to Program Status Register) 294
MOV (Move Word Data in Program Status Register to Destination Register)....338
Program Status (PS)....20

Program Status Register
ST (Store Word Data in Program Status Register to Memory)....392

PS
Program Status (PS)....20

R

R15
Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag)....75

Read-Modify-Write
Read-Modify-Write type Instructions 96

Recovery
Recovery from EIT Processing....45

Register
Timing of Reflection of Interrupt Level Mask Register (ILM) 64

Register Bypassing
Register Bypassing 74

Register Configuration
FR80 Family CPU Register Configuration 16

Register designated Field
Register designated Field....91

Register hazard
Occurrence of register hazard 74
Register hazards 74

Register Settings
Timing When Register Settings Are Reflected 63

Remain
DIV2 (Correction When Remain is 0) 177
DIV3 (Correction When Remain is 0) 179

Reset
Reset....42

RET
RET (Return from Subroutine) 366

RET:D
RET:D (Return from Subroutine)....368

RETI
RETI (Return from Interrupt) 370

Return
RET (Return from Subroutine) 366
RET:D (Return from Subroutine)....368
RETI (Return from Interrupt) 370

Return Pointer
Return Pointer (RP) 26

RP
Return Pointer (RP) 26

S

S flag
Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) 75

SCR
System Condition Code Register (SCR) 25

Search
SRCH0 (Search First Zero bit position distance From MSB) 373
SRCH1 (Search First One bit position distance From MSB) 375
SRCHC (Search First bit value change position distance From MSB)....377

Sign Extend
EXTSB (Sign Extend from Byte Data to Word Data) 221
EXTSH (Sign Extend from Byte Data to Word Data) 223

Signed Division
DIV0S (Initial Setting Up for Signed Division) 171
DIV4S (Correction Answer for Signed Division) 181
Signed Division....100

FR81 Family

Slot
Branching Instructions and Delay Slot 97
Software Interrupt
INT (Software Interrupt) 271
INTE (Software Interrupt for Emulator).............273
Source Register
ADD (Add Word Data of Source Register to Destination Register)...........................107
ADDC (Add Word Data of Source Register and Carry Bit to Destination Register)...................111
ADDN (Add Word Data of Source Register to Destination Register)...........................115
AND (And Word Data of Source Register to Data in Memory) 121
AND (And Word Data of Source Register to Destination Register)...........................123
ANDB (And Byte Data of Source Register to Data in Memory) 125
ANDH (And Halfword Data of Source Register to Data in Memory) 129
CMP (Compare Word Data in Source Register and Destination Register)...........................167
EOR (Exclusive Or Word Data of Source Register to Destination Register)...........................215
EOR (Exclusive Or Word Data of Source Register to Data in Memory) 213
EORB (Exclusive Or Byte Data of Source Register to Data in Memory) 217
EORH (Exclusive Or Halfword Data of Source Register to Data in Memory)................219
MOV (Move Word Data in Source Register to Destination Register)............334, 336, 340
MOV (Move Word Data in Source Register to Program Status Register) 342
OR (Or Word Data of Source Register to Data in Memory) 356
OR (Or Word Data of Source Register to Destination Register) 358
ORB (Or Byte Data of Source Register to Data in Memory) 360
ORH (Or Halfword Data of Source Register to Data in Memory) 364
SUB (Subtract Word Data in Source Register from Destination Register)...........................414
SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) 416
SUBN (Subtract Word Data in Source Register from Destination Register)...........................418
Special Usage
Special Usage of General-purpose Registers..........18
SRCH
SRCH0 (Search First Zero bit position distance From MSB) 373
SRCH1 (Search First One bit position distance From MSB) 375
SRCHC
SRCHC (Search First bit value change position distance From MSB)...........................377
SSP
System Stack Pointer (SSP)................................27
ST
ST (Store Word Data in Program Status Register to Memory) 392
ST (Store Word Data in Register to Memory) 379, 381, 383, 385, 387, 389, 390
Stack flag
Interlocking produced by reference to R15 and General-purpose Registers after Changing the Stack flag (S flag) 75
Stack Pointer
ADDSP (Add Stack Pointer and Immediate Data) 119
Relation between Stack Pointer and R15...............18
STB
STB (Store Byte Data in Register to Memory) 394, 396, 398, 400
Step Division Instructions
Step Division Instructions 100
Step Trace
Step Trace Traps 58
STH
STH (Store Halfword Data in Register to Memory) 401, 403, 405, 407
STILM
STILM (Set Immediate Data to Interrupt Level Mask Register)..408
STM
STM0 (Store Multiple Registers) 410
STM1 (Store Multiple Registers) 412
Store
ST (Store Word Data in Program Status Register to Memory) 392
ST (Store Word Data in Register to Memory) 379, 381, 383, 385, 387, 390
STB (Store Byte Data in Register to Memory) 394, 396, 398
STH (Store Halfword Data in Register to Memory) 401, 403, 405
STM0 (Store Multiple Registers) 410
STM1 (Store Multiple Registers) 412
SUB
SUB (Subtract Word Data in Source Register from Destination Register) 414

SUBC
SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) 416
SUBN
SUBN (Subtract Word Data in Source Register from Destination Register) 418
Subroutine
CALL (Call Subroutine) 157, 159
CALL:D (Call Subroutine) 161, 163
RET (Return from Subroutine) 366
RET:D (Return from Subroutine) 368
Subtract
SUB (Subtract Word Data in Source Register from Destination Register) 414
SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) 416
SUBN (Subtract Word Data in Source Register from Destination Register) 418
System Condition Code Register
System Condition Code Register (SCR) 25
System Stack Pointer
System Stack Pointer (SSP) 27
System Status Register
System Status Register (SSR) 21

T

Table Base Register
Table Base Register (TBR) 29
Test
BTSTH (Test Higher 4bit of Byte Data in Memory) 153
BTSTL (Test Lower 4bit of Byte Data in Memory) 155
Trace
Step Trace Traps 58
Traps
Step Trace Traps 58
Traps 57
TBR
Table Base Register (TBR) 29

U

Unsign Extend
EXTUB (Unsign Extend from Byte Data to Word Data) 225
EXTUH (Unsign Extend from Byte Data to Word Data) 227
Unsigned Division
DIV0U (Initial Setting Up for Unsigned Division) 173
Unsigned Division 101
Unsigned Halfword Data
MULUH (Multiply Unsigned Halfword Data) 352
Unsigned Word Data
MULU (Multiply Unsigned Word Data) 350
User Interrupt
Points of Caution while using User Interrupts 66
Preparation while using user interrupts 65
Usage Sequence of User Interrupts 65
User Stack Pointer
User Stack Pointer (USP) 28
USP
User Stack Pointer (USP) 28

V

value change
SRCHC (Search First bit value change position distance From MSB) 377
Vector Table
Vector Table Area 9

W

Word Alignment
Word Alignment 14
Word Data
ADD (Add Word Data of Source Register to Destination Register) 107
ADDC (Add Word Data of Source Register and Carry Bit to Destination Register) 111
ADDN (Add Word Data of Source Register to Destination Register) 115
AND (And Word Data of Source Register to Data in Memory) 121
AND (And Word Data of Source Register to Destination Register) 123
CMP (Compare Word Data in Source Register and Destination Register) 167
DMOV (Move Word Data from Direct Address to Post Increment Register Indirect Address) 187
DMOV (Move Word Data from Direct Address to Pre Decrement Register Indirect Address) 191
DMOV (Move Word Data from Direct Address to Register) 183
DMOV (Move Word Data from Post Increment Register Indirect Address to Direct Address) 189, 193
DMOV (Move Word Data from Register to Direct Address) 185
EOR (Exclusive Or Word Data of Source Register to Destination Register) 215

EOR (Exclusive Or Word Data of Source Register to Data in Memory) 213
EXTSB (Sign Extend from Byte Data to Word Data) 221
EXTSH (Sign Extend from Byte Data to Word Data) 223
EXTUB (Unsign Extend from Byte Data to Word Data) 225
EXTUH (Unsign Extend from Byte Data to Word Data) 227
LD (Load Word Data in Memory to Program Status Register) 294
LD (Load Word Data in Memory to Register) 281, 283, 285, 287, 289, 292
MOV (Move Word Data in Program Status Register to Destination Register)...........................338
MOV (Move Word Data in Source Register to Destination Register)............334, 336, 340
MOV (Move Word Data in Source Register to Program Status Register) 342
MUL (Multiply Word Data) 346
MULU (Multiply Unsigned Word Data) 350
OR (Or Word Data of Source Register to Data in Memory)..356
OR (Or Word Data of Source Register to Destination Register)..358
ST (Store Word Data in Program Status Register to Memory) 392
ST (Store Word Data in Register to Memory) 379, 381, 383, 385, 387, 390
SUB (Subtract Word Data in Source Register from Destination Register) 414
SUBC (Subtract Word Data in Source Register and Carry bit from Destination Register) 416
SUBN (Subtract Word Data in Source Register from Destination Register) 418
Word Data 12

X

XCHB
XCHB (Exchange Byte Data)............................420

CM71-00105-1E

FUJITSU SEMICONDUCTOR • CONTROLLER MANUAL

FR81 Family

32-BIT MICROCONTROLLER

PROGRAMMING MANUAL

August 2009 the first edition

Published **FUJITSU MICROELECTRONICS LIMITED**

Edited Sales Promotion Dept.